Introduction to “hsdar“

Hanna Meyer, Lukas W. Lehnert

September 9, 2016

Contents

1 Introduction

This tutorial introduces techniques for creating, handling, manipulating, analyzing and simulating hyperspectral data using the hsdar package. Though we'll try our best to keep the examples as easy as possible, we assume that you are already familiar with the R software.

In most cases this tutorial is not built successively so that you can get in at your chapter of interested. However, you should briefly run over this introduction to see how to use this tutorial.

The tutorial focuses on the usage of hsdar for the calculation of several common methods in hyperspectral data manipulation and analysis. Despite some less common methods, we won't explain in detail what the methods do and in which cases they are useful or not. Please read the hsdar help files and references for more information about your methods of interest.

hsdar is still experimental. If you have any questions, suggestions or concerns don't hesitate to contact the authors. For some applications of hsdar see Lehnert et al. (2013, 2014, 2015); Meyer et al. (2013).

1.1 Sample data

Almost all of the exercises in this tutorial use one single sample dataset - "spectral_data" - which is included in the hsdar package. This dataset was created on a free air enrichment site (FACE) near Giessen, central Germany.

In the first line, the dataset contains hyperspectral reflectance measurements which were taken with a field spectrometer from a height of approx. 1.50 m covering the integrated spectrum of a circle of approx. 50 cm on the ground. Thus, reflectance values result from different fractions of vegetation, soil, stones etc. Furthermore, the dataset contains chlorophyll content of the vegetation.

1.2 How to start

To work with the tutorial, first install the hsdar package and load the library as well as the sample data:

```
> #install.packages("hsdar")
> library(hsdar)
> data(spectral_data) #Load the data used in the tutorial
```

If you need help, see

```
> help(hsdar)
```

2 Handling of speclibs

In this chapter a "Speclib" which is the main class of hsdar will be presented. Almost all functions of hsdar require that your Spectra are stored in a Speclib. To learn how to handle Speclibs, we will first have a look on the structure of the sample data. Afterwards it will be shown how to create own Speclibs and how to read and write them.

2.1 Structure

Hyperspectral data as well as further information related to these data are stored in a class called ”Speclib”. To understand the structure of a Speclib, have a look on the sample Speclib ”spectral_data”

```
> spectral_data #See how Speclibs are printed

Summary of Speclib


History of usage
---------------------
(1)   Reflectance = mean applied to matrix spectra by attribute 'site'


Summary of spectra
---------------------
Total number of spectra : 45
Number of bands : 1401
Width of bands : 1
```

The printed information of a Speclib contain the number of spectra, the number of spectral bands and the width of the bands. However, there are more information stored in the Speclib. Have a look on the structure of ”spectral_data” to see all its information:

```
> str(spectral_data)

Formal class 'Speclib' [package "hsdar"] with 13 slots
  ..@ spectra           :Formal class '.Spectra' [package "hsdar"] with 3 slots
  .. .. ..@ fromRaster: logi FALSE
  .. .. ..@ spectra_ma: num [1:45, 1:1401] 6.52 7.01 7.25 6.74 7.52 ...
  .. .. ..@ spectra_ra:Formal class 'RasterBrick' [package "raster"] with 12 slots
  .. .. .. .. ..@ file    :Formal class '.RasterFile' [package "raster"] with 13 slots
  .. .. .. .. .. .. ..@ name        : chr ""
  .. .. .. .. .. .. ..@ datanotation: chr "FLT4S"
  .. .. .. .. .. .. ..@ byteorder   : chr "little"
  .. .. .. .. .. .. ..@ nodatavalue : num -Inf
  .. .. .. .. .. .. ..@ NAchanged   : logi FALSE
  .. .. .. .. .. .. ..@ nbands      : int 1
  .. .. .. .. .. .. ..@ bandorder   : chr "BIL"
  .. .. .. .. .. .. ..@ offset      : int 0
  .. .. .. .. .. .. ..@ toptobottom : logi TRUE
  .. .. .. .. .. .. ..@ blockrows   : int 0
  .. .. .. .. .. .. ..@ blockcols   : int 0
  .. .. .. .. .. .. ..@ driver      : chr ""
  .. .. .. .. .. .. ..@ open        : logi FALSE
  .. .. .. .. ..@ data    :Formal class '.MultipleRasterData' [package "raster"] with 14 slots
  .. .. .. .. .. .. ..@ values    : logi[0 , 0 ]
  .. .. .. .. .. .. ..@ offset    : num 0
  .. .. .. .. .. .. ..@ gain      : num 1
  .. .. .. .. .. .. ..@ inmemory  : logi FALSE
  .. .. .. .. .. .. ..@ fromdisk  : logi FALSE
  .. .. .. .. .. .. ..@ nlayers   : int 0
  .. .. .. .. .. .. ..@ dropped   : NULL
  .. .. .. .. .. .. ..@ isfactor  : logi FALSE
  .. .. .. .. .. .. ..@ attributes: list()
  .. .. .. .. .. .. ..@ haveminmax: logi FALSE
  .. .. .. .. .. .. ..@ min       : num Inf
  .. .. .. .. .. .. ..@ max       : num -Inf
  .. .. .. .. .. .. ..@ unit      : chr ""
  .. .. .. .. .. .. ..@ names     : chr ""
  .. .. .. .. ..@ legend  :Formal class '.RasterLegend' [package "raster"] with 5 slots
  .. .. .. .. .. .. ..@ type      : chr(0)
  .. .. .. .. .. .. ..@ values    : logi(0)
  .. .. .. .. .. .. ..@ color     : logi(0)
  .. .. .. .. .. .. ..@ names     : logi(0)
  .. .. .. .. .. .. ..@ colortable: logi(0)
  .. .. .. .. ..@ title   : chr(0)
  .. .. .. .. ..@ extent  :Formal class 'Extent' [package "raster"] with 4 slots
  .. .. .. .. .. .. ..@ xmin: num 0
  .. .. .. .. .. .. ..@ xmax: num 1
  .. .. .. .. .. .. ..@ ymin: num 0
  .. .. .. .. .. .. ..@ ymax: num 1
  .. .. .. .. ..@ rotated : logi FALSE
  .. .. .. .. ..@ rotation:Formal class '.Rotation' [package "raster"] with 2 slots
  .. .. .. .. .. .. ..@ geotrans: num(0)
  .. .. .. .. .. .. ..@ transfun:function ()
  .. .. .. .. ..@ ncols   : int 1
  .. .. .. .. ..@ nrows   : int 1
  .. .. .. .. ..@ crs     :Formal class 'CRS' [package "sp"] with 1 slot
  .. .. .. .. .. .. ..@ projargs: chr NA
  .. .. .. .. ..@ history : list()
  .. .. .. .. ..@ z       : list()
  ..@ wavelength         : num [1:1401] 305 306 307 308 309 310 311 312 313 314 ...
  ..@ attributes         :'data.frame':       45 obs. of  4 variables:
  .. ..$ year       : num [1:45] 2014 2014 2014 2014 2014 ...
  .. ..$ season     : chr [1:45] "summer" "summer" "summer" "summer" ...
  .. ..$ site       : chr [1:45] "C1" "C2" "C3" "K1" ...
  .. ..$ chlorophyll: num [1:45] 25.2 23.7 31.5 19.1 23.3 ...
  ..@ fwhm               : num 1
  ..@ continuousdata     : logi TRUE
  ..@ wlunit             : chr "nm"
  ..@ xlabel             : chr "Wavelength"
  ..@ ylabel             : chr "Reflectance"
  ..@ ID                 : chr [1:45] "1" "2" "3" "4" ...
  ..@ wavelength.is.range: logi FALSE
  ..@ transformation     : chr(0)
  ..@ usagehistory       : chr "Reflectance = mean applied to matrix spectra by attribute 'site'"
  ..@ rastermeta         : list()
```

Only considering the most important components (slots), the Speclib contains spectra, wavelengths, different attributes (optional) and some metadata (optional). The spectra are stored in a matrix with the spectral bands organized in columns and the different samples (or pixels) organized in rows. The vector ”wavelength” indicates the corresponding wavelength of each band. The ”attributes” data.frame offers a possibility to store further information related to the spectra, in this case, these are ”location”, ”ID” and several environmental variables like e.g., the soil moisture. Thus, the wavelength contains the metadata of the column and the attributes contain the information of the rows of the spectra-matrix. Finally, some metadata are given like ”reflectance” as the type of the spectra or ”nm” as the unit of the wavelength.

2.2 Creating Speclibs

Now, we will explain how to create your own Speclibs. We will do this the way that we split the sample Speclib back into its components and then show how to bring the components together into a new Speclib.

2.2.1 Speclibs from matrices

Speclibs can be created in different ways. To build a Speclib you need at least spectra and the corresponding wavelength values. The easiest way is to prepare a matrix of your spectra. This matrix must be organized in the way that each row represents one sample and each column represents a spectral band. To go on with the example introduced above, we transform the spectra of ”spectral_data back into a matrix:

```
> spectra <- spectra(spectral_data)
```

See what happened:

```
> str(spectra)

 num [1:45, 1:1401] 6.52 7.01 7.25 6.74 7.52 ...
```

This is how the input matrix must look like: We have a matrix with each row representing one spectrum and each column representing one channel. Further, we need a vector indicating which wavelength corresponds to each column. Therefore we will extract the wavelength from "spectral_data":

```
> wavelength <- wavelength(spectral_data)
```

Now both components needed to create a Speclib are available: spectra and the corresponding wavelength. Now you can build a new Speclib from them:

```
> newSpeclib <- speclib(spectra, wavelength)
```

Having a look at the structure showing that you have re-created "spectral_data":

```
> str(newSpeclib)

Formal class 'Speclib' [package "hsdar"] with 13 slots
  ..@ spectra           :Formal class '.Spectra' [package "hsdar"] with 3 slots
  .. .. ..@ fromRaster: logi FALSE
  .. .. ..@ spectra_ma: num [1:45, 1:1401] 6.52 7.01 7.25 6.74 7.52 ...
  .. .. ..@ spectra_ra:Formal class 'RasterBrick' [package "raster"] with 12 slots
  .. .. .. .. ..@ file    :Formal class '.RasterFile' [package "raster"] with 13 slots
  .. .. .. .. .. .. ..@ name        : chr ""
  .. .. .. .. .. .. ..@ datanotation: chr "FLT4S"
  .. .. .. .. .. .. ..@ byteorder   : chr "little"
  .. .. .. .. .. .. ..@ nodatavalue : num -Inf
  .. .. .. .. .. .. ..@ NAchanged   : logi FALSE
  .. .. .. .. .. .. ..@ nbands      : int 1
  .. .. .. .. .. .. ..@ bandorder   : chr "BIL"
  .. .. .. .. .. .. ..@ offset      : int 0
  .. .. .. .. .. .. ..@ toptobottom : logi TRUE
  .. .. .. .. .. .. ..@ blockrows   : int 0
  .. .. .. .. .. .. ..@ blockcols   : int 0
  .. .. .. .. .. .. ..@ driver      : chr ""
  .. .. .. .. .. .. ..@ open        : logi FALSE
  .. .. .. .. ..@ data    :Formal class '.MultipleRasterData' [package "raster"] with 14 slots
  .. .. .. .. .. .. ..@ values    : logi[0 , 0 ]
  .. .. .. .. .. .. ..@ offset    : num 0
  .. .. .. .. .. .. ..@ gain      : num 1
  .. .. .. .. .. .. ..@ inmemory  : logi FALSE
  .. .. .. .. .. .. ..@ fromdisk  : logi FALSE
  .. .. .. .. .. .. ..@ nlayers   : int 0
  .. .. .. .. .. .. ..@ dropped   : NULL
  .. .. .. .. .. .. ..@ isfactor  : logi FALSE
  .. .. .. .. .. .. ..@ attributes: list()
  .. .. .. .. .. .. ..@ haveminmax: logi FALSE
  .. .. .. .. .. .. ..@ min       : num Inf
  .. .. .. .. .. .. ..@ max       : num -Inf
  .. .. .. .. .. .. ..@ unit      : chr ""
  .. .. .. .. .. .. ..@ names     : chr ""
  .. .. .. .. ..@ legend  :Formal class '.RasterLegend' [package "raster"] with 5 slots
  .. .. .. .. .. .. ..@ type      : chr(0)
  .. .. .. .. .. .. ..@ values    : logi(0)
  .. .. .. .. .. .. ..@ color     : logi(0)
  .. .. .. .. .. .. ..@ names     : logi(0)
  .. .. .. .. .. .. ..@ colortable: logi(0)
  .. .. .. .. ..@ title   : chr(0)
  .. .. .. .. ..@ extent  :Formal class 'Extent' [package "raster"] with 4 slots
  .. .. .. .. .. .. ..@ xmin: num 0
  .. .. .. .. .. .. ..@ xmax: num 1
  .. .. .. .. .. .. ..@ ymin: num 0
  .. .. .. .. .. .. ..@ ymax: num 1
  .. .. .. .. ..@ rotated : logi FALSE
  .. .. .. .. ..@ rotation:Formal class '.Rotation' [package "raster"] with 2 slots
  .. .. .. .. .. .. ..@ geotrans: num(0)
  .. .. .. .. .. .. ..@ transfun:function ()
  .. .. .. .. ..@ ncols   : int 1
  .. .. .. .. ..@ nrows   : int 1
  .. .. .. .. ..@ crs     :Formal class 'CRS' [package "sp"] with 1 slot
  .. .. .. .. .. .. ..@ projargs: chr NA
  .. .. .. .. ..@ history : list()
  .. .. .. .. ..@ z       : list()
  ..@ wavelength        : num [1:1401] 305 306 307 308 309 310 311 312 313 314 ...
  ..@ attributes        :'data.frame':       0 obs. of  0 variables
  ..@ fwhm              : num 1
  ..@ continuousdata    : logi TRUE
  ..@ wlunit            : chr "nm"
  ..@ xlabel            : chr "Wavelength"
  ..@ ylabel            : chr "Reflectance"
  ..@ ID                : chr(0)
  ..@ wavelength.is.range: logi FALSE
  ..@ transformation    : chr(0)
  ..@ usagehistory      : chr(0)
  ..@ rastermeta        : list()
```

However, it would be nice to have an ID for each spectrum:

```
> ids <- idSpeclib(spectral_data) #extract ID from "spectral_data"
> idSpeclib(newSpeclib) <- as.character(ids) #...and assign them to the
>                                             #new Speclib
```

Still the attributes are missing in the new Speclib. Those can be handled with the function ”attribute”: If you extract the attributes from ”spectral_data” using ”attribute”, you get a data.frame containing the values for each attribute at each site:

```
> attributes <- attribute(spectral_data)
> head(attributes)

  year season site chlorophyll
1 2014 summer   C1    25.18261
2 2014 summer   C2    23.65696
3 2014 summer   C3    31.50000
```

```
4 2014 summer   K1    19.12000
5 2014 summer   K2    23.31818
6 2014 summer   K3    25.21500
```

You can now use this data.frame to complete your new Speclib:

```
> attribute(newSpeclib) <- attributes
```

Finally you have a Speclib which is well comparable to the exemplary Speclib:

```
> str(newSpeclib)

Formal class 'Speclib' [package "hsdar"] with 13 slots
  ..@ spectra           :Formal class '.Spectra' [package "hsdar"] with 3 slots
  .. .. ..@ fromRaster: logi FALSE
  .. .. ..@ spectra_ma: num [1:45, 1:1401] 6.52 7.01 7.25 6.74 7.52 ...
  .. .. ..@ spectra_ra:Formal class 'RasterBrick' [package "raster"] with 12 slots
  .. .. .. .. ..@ file    :Formal class '.RasterFile' [package "raster"] with 13 slots
  .. .. .. .. .. .. ..@ name        : chr ""
  .. .. .. .. .. .. ..@ datanotation: chr "FLT4S"
  .. .. .. .. .. .. ..@ byteorder   : chr "little"
  .. .. .. .. .. .. ..@ nodatavalue : num -Inf
  .. .. .. .. .. .. ..@ NAchanged   : logi FALSE
  .. .. .. .. .. .. ..@ nbands      : int 1
  .. .. .. .. .. .. ..@ bandorder   : chr "BIL"
  .. .. .. .. .. .. ..@ offset      : int 0
  .. .. .. .. .. .. ..@ toptobottom : logi TRUE
  .. .. .. .. .. .. ..@ blockrows   : int 0
  .. .. .. .. .. .. ..@ blockcols   : int 0
  .. .. .. .. .. .. ..@ driver      : chr ""
  .. .. .. .. .. .. ..@ open        : logi FALSE
  .. .. .. .. ..@ data    :Formal class '.MultipleRasterData' [package "raster"] with 14 slots
  .. .. .. .. .. .. ..@ values    : logi[0 , 0 ]
  .. .. .. .. .. .. ..@ offset    : num 0
  .. .. .. .. .. .. ..@ gain      : num 1
  .. .. .. .. .. .. ..@ inmemory  : logi FALSE
  .. .. .. .. .. .. ..@ fromdisk  : logi FALSE
  .. .. .. .. .. .. ..@ nlayers   : int 0
  .. .. .. .. .. .. ..@ dropped   : NULL
  .. .. .. .. .. .. ..@ isfactor  : logi FALSE
  .. .. .. .. .. .. ..@ attributes: list()
  .. .. .. .. .. .. ..@ haveminmax: logi FALSE
  .. .. .. .. .. .. ..@ min       : num Inf
  .. .. .. .. .. .. ..@ max       : num -Inf
  .. .. .. .. .. .. ..@ unit      : chr ""
  .. .. .. .. .. .. ..@ names     : chr ""
  .. .. .. .. ..@ legend  :Formal class '.RasterLegend' [package "raster"] with 5 slots
  .. .. .. .. .. .. ..@ type      : chr(0)
  .. .. .. .. .. .. ..@ values    : logi(0)
  .. .. .. .. .. .. ..@ color     : logi(0)
  .. .. .. .. .. .. ..@ names     : logi(0)
  .. .. .. .. .. .. ..@ colortable: logi(0)
  .. .. .. .. ..@ title   : chr(0)
  .. .. .. .. ..@ extent  :Formal class 'Extent' [package "raster"] with 4 slots
  .. .. .. .. .. .. ..@ xmin: num 0
  .. .. .. .. .. .. ..@ xmax: num 1
  .. .. .. .. .. .. ..@ ymin: num 0
  .. .. .. .. .. .. ..@ ymax: num 1
  .. .. .. .. ..@ rotated : logi FALSE
  .. .. .. .. ..@ rotation:Formal class '.Rotation' [package "raster"] with 2 slots
  .. .. .. .. .. .. ..@ geotrans: num(0)
  .. .. .. .. .. .. ..@ transfun:function ()
  .. .. .. .. ..@ ncols   : int 1
  .. .. .. .. ..@ nrows   : int 1
  .. .. .. .. ..@ crs     :Formal class 'CRS' [package "sp"] with 1 slot
  .. .. .. .. .. .. ..@ projargs: chr NA
  .. .. .. .. ..@ history : list()
  .. .. .. .. ..@ z       : list()
  ..@ wavelength         : num [1:1401] 305 306 307 308 309 310 311 312 313 314 ...
  ..@ attributes         :'data.frame':        45 obs. of  4 variables:
  .. ..$ year       : num [1:45] 2014 2014 2014 2014 2014 ...
  .. ..$ season     : chr [1:45] "summer" "summer" "summer" "summer" ...
  .. ..$ site       : chr [1:45] "C1" "C2" "C3" "K1" ...
  .. ..$ chlorophyll: num [1:45] 25.2 23.7 31.5 19.1 23.3 ...
  ..@ fwhm               : num 1
  ..@ continuousdata     : logi TRUE
  ..@ wlunit             : chr "nm"
  ..@ xlabel             : chr "Wavelength"
  ..@ ylabel             : chr "Reflectance"
  ..@ ID                 : chr [1:45] "1" "2" "3" "4" ...
  ..@ wavelength.is.range: logi FALSE
  ..@ transformation     : chr(0)
  ..@ usagehistory       : chr(0)
  ..@ rastermeta         : list()
```

2.2.2 Speclibs from raster files

The investigation of data taken with a hyperspectral camera or by hyperspectral satellite sensors (e.g., Hyperion) brings along that a huge number of spectra must be analyzed. Unless you are working on a large cluster, R won't be able to store all of the data in RAM in this case. A workaround is, to analyze each row of the image separately. This is possible as far as you are using techniques which do not take the neighboring spectra into account. This is the case for almost all functions in this tutorial above except the correlation and linear regression techniques. A very good starting point for the row-wise analysis of large raster files is given in the tutorials of the ”raster”-package the hsdar-package is depending on. Nevertheless, there are two specific difficulties if hyperspectral data should be analyzed:

1. The wavelength information must be stored along the spectra
2. Most of the function in the hsdar-package require that data is transferred to functions as Speclib.

Thus, a small extension of the raster-classes is provided by the hsdar-package: the class HyperSpecRaster. This class is more or less a RasterBrick-object with wavelength information.

In the following example which is taken from the help page of HyperSpecRaster, we will first create a small hyperspectral raster file using PROSAIL (for explanation of PROSAIL see section 4):

```
> ## Create raster file using PROSAIL
> ## Run PROSAIL
> parameter <- data.frame(N = c(rep.int(seq(0.5, 1.4, 0.1), 6)),
+                         LAI = c(rep.int(0.5, 10), rep.int(1, 10),
+                                 rep.int(1.5, 10), rep.int(2, 10),
+                                 rep.int(2.5, 10), rep.int(3, 10)))
> spectra <- PROSAIL(parameterList = parameter)
> ## Create SpatialPixelsDataFrame and fill data with spectra from
> ## PROSAIL
> rows <- round(nspectra(spectra)/10, 0)
> cols <- ceiling(nspectra(spectra)/rows)
> grd <- SpatialGrid(GridTopology(cellcentre.offset = c(1,1,1),
+                                 cellsize = c(1,1,1),
+                                 cells.dim = c(cols, rows, 1)))
> x <- SpatialPixelsDataFrame(grd,
+                             data = as.data.frame(spectra(spectra)))
> ## Write data to example file (example_in.tif) in workingdirectory
> writeGDAL(x, fname = "example_in.tif", drivername = "GTiff")
```

Once the file is created, we will read it back into R and create an object of class HyperSpecRaster from it:

```
> infile <- "example_in.tif"
> wavelength <- spectra$wavelength
> ra <- HyperSpecRaster(infile, wavelength)
> tr <- blockSize(ra)
```

Note that we haven't read the values of the file into memory, yet. This is now performed in a small loop over all rows: Let's assume that we want to calculate all available vegetation indices from the hyperspectral image. Thus, we will read each row into memory, calculate the vegetation indices from the subset of pixels in the memory and store the output in a new file:

```
> outfile <- "example_result.tif"
> n_veg <- as.numeric(length(vegindex()))
> res <- writeStart(ra, outfile, overwrite = TRUE, nl = n_veg)
> for (i in 1:tr$n)
+ {
+   v <- getValuesBlock(ra, row=tr$row[i], nrows=tr$nrows[i])
+   mask(v) <- c(1350, 1450)
+   v <- as.matrix(vegindex(v, index=vegindex()))
+   res <- writeValues(res, v, tr$row[i])
+ }
> res <- writeStop(res)
```

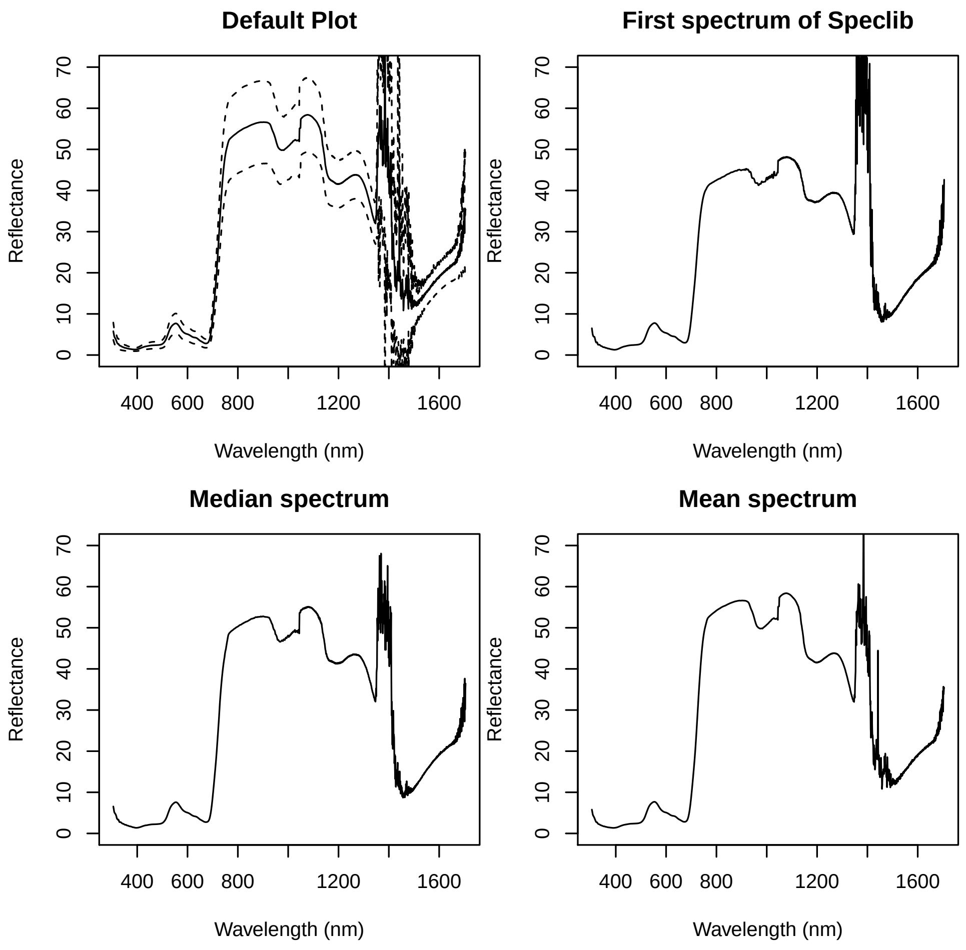
Note that hsdar is automatically transferring the data to an object of class Speclib during each step in the loop (so, `v` is a Speclib).

Depending on the number of columns in your image and on the amount of memory of your computer the loop may significantly speed up if you read multiple rows per iteration step. See the tutorial in the raster package mentioned above for further examples and information.

3 Plotting Speclibs

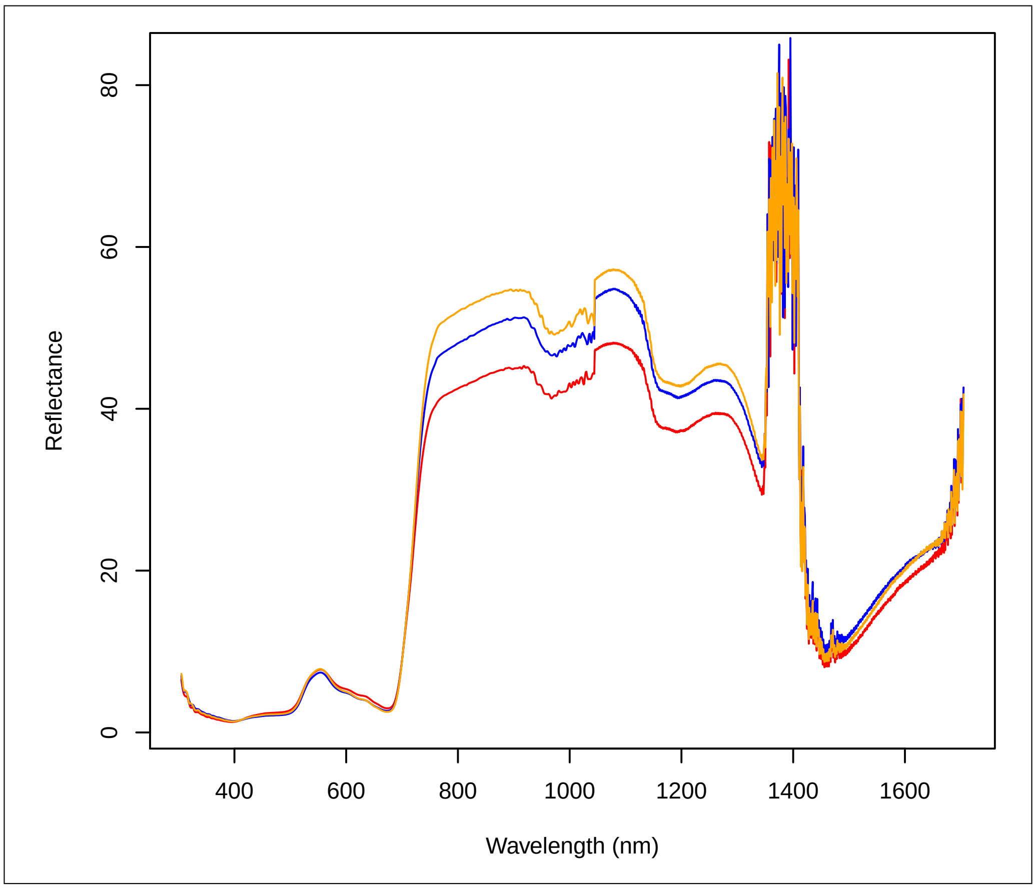
Speclibs can easily be plotted using the plot.speclib function. The default way is to plot mean values (solid line) of all spectra in the Speclib and the standard deviations within bands. If the data are continuous the standard deviations are plotted as dashed lines otherwise error bars will indicate standard deviations. You can also plot single spectra by adapting the FUN parameter to the ID of the spectra to be plotted. Also, you can use a function as FUN parameter like e.g. the median or mean spectrum. See some examples below:

```
> par(mfrow = c(2,2))
> plot(spectral_data, main = "Default Plot")
> plot(spectral_data, FUN = 1, main = "First spectrum of Speclib")
> plot(spectral_data, FUN = "median", main = "Median spectrum")
> plot(spectral_data, FUN = "mean", main = "Mean spectrum")
```



There are some more parameters which might be interesting to plot the data: The parameter "new" allows you to plot more than one spectrum in one plot:

```
> plot(spectral_data, FUN = 1, col = "red")
> plot(spectral_data, FUN = 2, col = "blue", new = FALSE)
> plot(spectral_data, FUN = 3, col = "orange", new = FALSE)
```

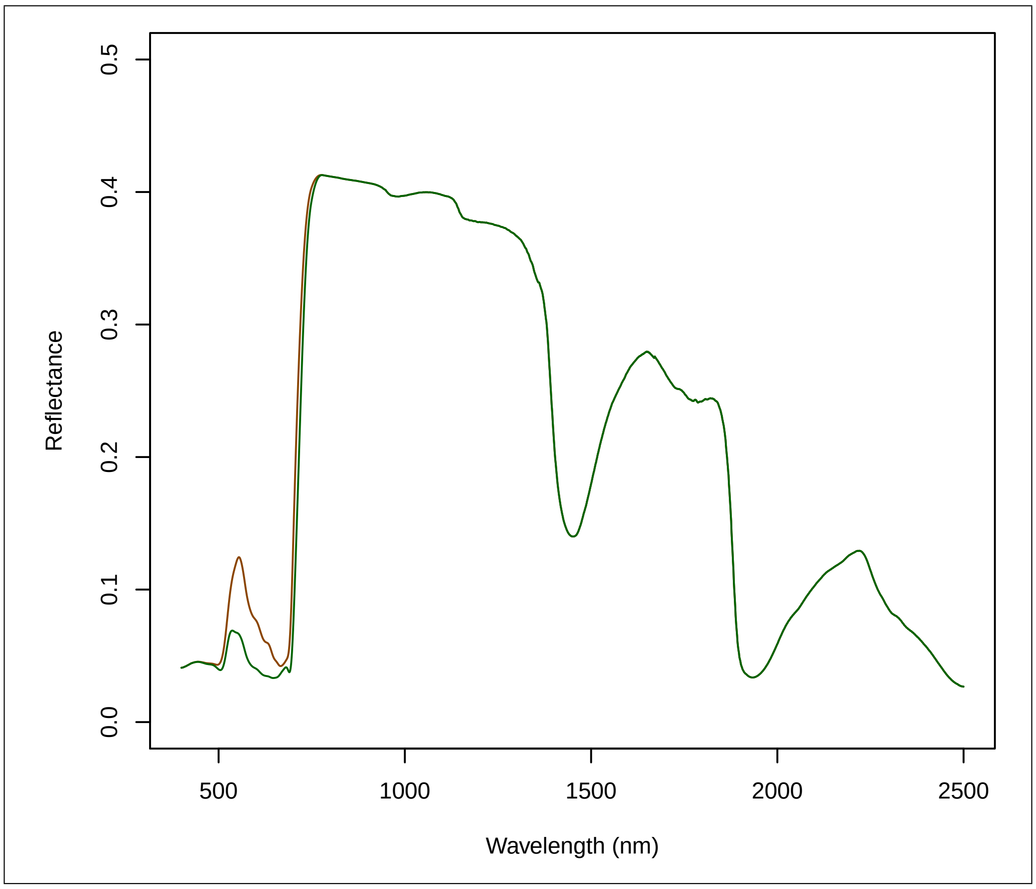


Beside these specific arguments to plot spectra, any arguments known from the default plot function can be used.

4 Simulating spectra with PROSPECT and PROSAIL

PROSPECT is a widely used leaf reflectance (or transmittance) model which simulates reflectance values between 400 and 2500 nm. For a detailed description of PROSPECT see Jacquemoud and Baret (1990). PROSPECT requires a set of parameters describing structure and chemical composition of leafs. In hsdar there are default values for each parameter. However, these default values were included with the intention to provide an easy access to the model and should be used with care in any scientific approach! But now, let's jump into the simulation of reflectance values:

```
> ## Simulate first spectrum with lower chlorophyll content
> spectrum1 <- PROSPECT(N = 1.3, Cab = 30, Car = 10, Cbrown = 0,
+                       Cw = 0.01, Cm = 0.01)
> ## Simulate second spectrum with higher chlorophyll content
> spectrum2 <- PROSPECT(N = 1.3, Cab = 60, Car = 10, Cbrown = 0,
+                       Cw = 0.01, Cm = 0.01)
> ## Plot results:
> plot(spectrum1, col = "darkorange4", ylim = c(0,0.5),
+      subset = c(400, 800))
> plot(spectrum2, col = "darkgreen", new = FALSE)
```

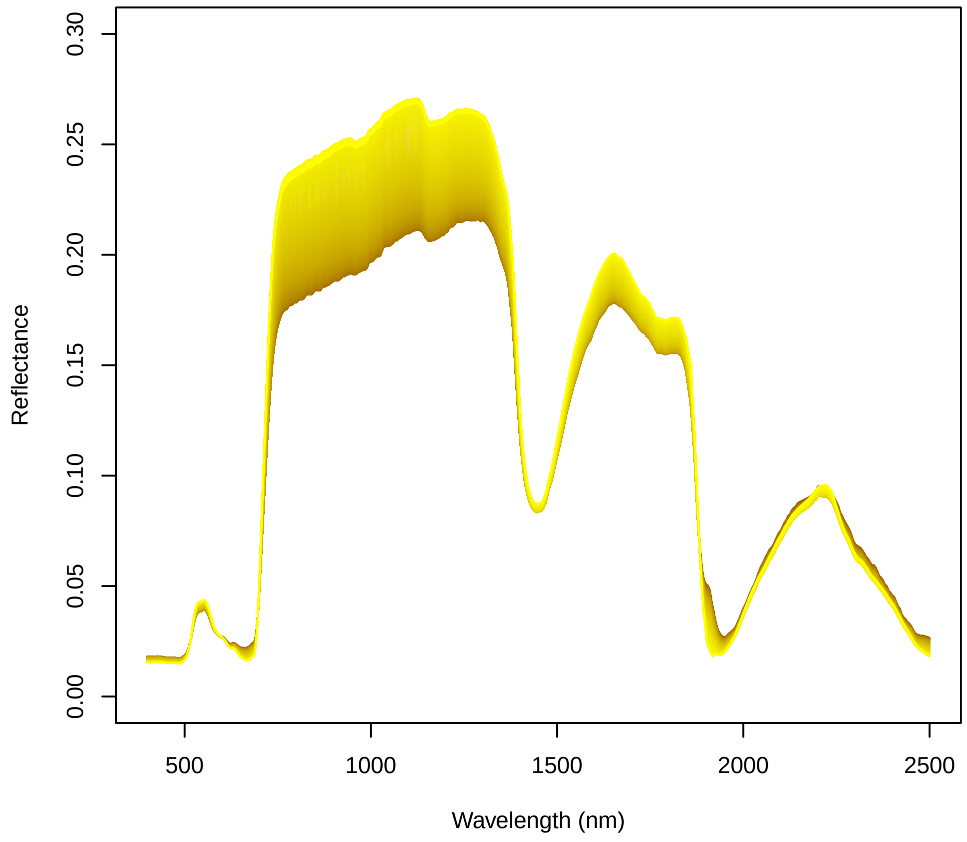


In addition to PROSPECT, PROSAIL simulates the reflectance of the canopy of vegetation. Thus, the number of parameters is considerably larger and includes the geometry of the plants and the viewing and illumination geometry. General information about PROSAIL may be found in Jacquemoud et al. (2009). In the following example, we will use another way to specify the parameters which is available in PROSPECT and PROSAIL: the parameterList. Let's assume we want to test the effect of the illumination geometry (the solar zenith angle) on reference values. It would be possible to simulate many different spectra with different parameter settings. However, we would then have a confusing number of speclibs. An easier way is the parameterList (which is a data.frame in R):

```
> ## Defining parameter
> parameter <- data.frame(tts = seq(15, 85, 0.5))
> head(parameter)

   tts
1 15.0
2 15.5
3 16.0
4 16.5
5 17.0
6 17.5

> ## Perform simulation (all other parameters are set to default
> ## values)
> spectra <- PROSAIL(parameterList = parameter)
> spectra

Summary of Speclib


Summary of spectra
---------------------
Total number of spectra : 141
Number of bands : 2101
Width of bands : 1

> ## Let's see the attributes
> summary(spectra$attributes)

       N             Cab          Car         Cbrown        Cw
 Min.   :1.5   Min.   :40   Min.   :8   Min.   :0   Min.   :0.01
 1st Qu.:1.5   1st Qu.:40   1st Qu.:8   1st Qu.:0   1st Qu.:0.01
 Median :1.5   Median :40   Median :8   Median :0   Median :0.01
 Mean   :1.5   Mean   :40   Mean   :8   Mean   :0   Mean   :0.01
 3rd Qu.:1.5   3rd Qu.:40   3rd Qu.:8   3rd Qu.:0   3rd Qu.:0.01
 Max.   :1.5   Max.   :40   Max.   :8   Max.   :0   Max.   :0.01
       Cm             psoil        LAI       TypeLidf     lidfa
 Min.   :0.009   Min.   :0   Min.   :1   Min.   :1   Min.   :-0.35
 1st Qu.:0.009   1st Qu.:0   1st Qu.:1   1st Qu.:1   1st Qu.:-0.35
 Median :0.009   Median :0   Median :1   Median :1   Median :-0.35
 Mean   :0.009   Mean   :0   Mean   :1   Mean   :1   Mean   :-0.35
 3rd Qu.:0.009   3rd Qu.:0   3rd Qu.:1   3rd Qu.:1   3rd Qu.:-0.35
 Max.   :0.009   Max.   :0   Max.   :1   Max.   :1   Max.   :-0.35
     lidfb           hspot           tts            tto
 Min.   :-0.15   Min.   :0.01   Min.   :15.0   Min.   :10
 1st Qu.:-0.15   1st Qu.:0.01   1st Qu.:32.5   1st Qu.:10
 Median :-0.15   Median :0.01   Median :50.0   Median :10
 Mean   :-0.15   Mean   :0.01   Mean   :50.0   Mean   :10
 3rd Qu.:-0.15   3rd Qu.:0.01   3rd Qu.:67.5   3rd Qu.:10
 Max.   :-0.15   Max.   :0.01   Max.   :85.0   Max.   :10
      psi
 Min.   :0
 1st Qu.:0
 Median :0
 Mean   :0
 3rd Qu.:0
 Max.   :0
```

We can visualize the effect of the solar zenith angle simply plotting the spectra in different colours:

```
> colours <- colorRamp(c("darkorange4", "yellow"))
> plot(spectra, FUN = 1, ylim = c(0, 0.3),
+      col = rgb(colours(spectra$attributes$tts[1]/85),
+                 maxColorValue = 255))
> for (i in 2:nspectra(spectra))
+   plot(spectra, FUN = i, new = FALSE,
+        col = rgb(colours(spectra$attributes$tts[i]/85),
+                  maxColorValue = 255))
```



Of course it is also possible to test the effect of two parameters on the reflectance values. In the following example we will plot the reflectance values of canopies with three different LAI values. Within each LAI class we vary the leaf angle distribution:

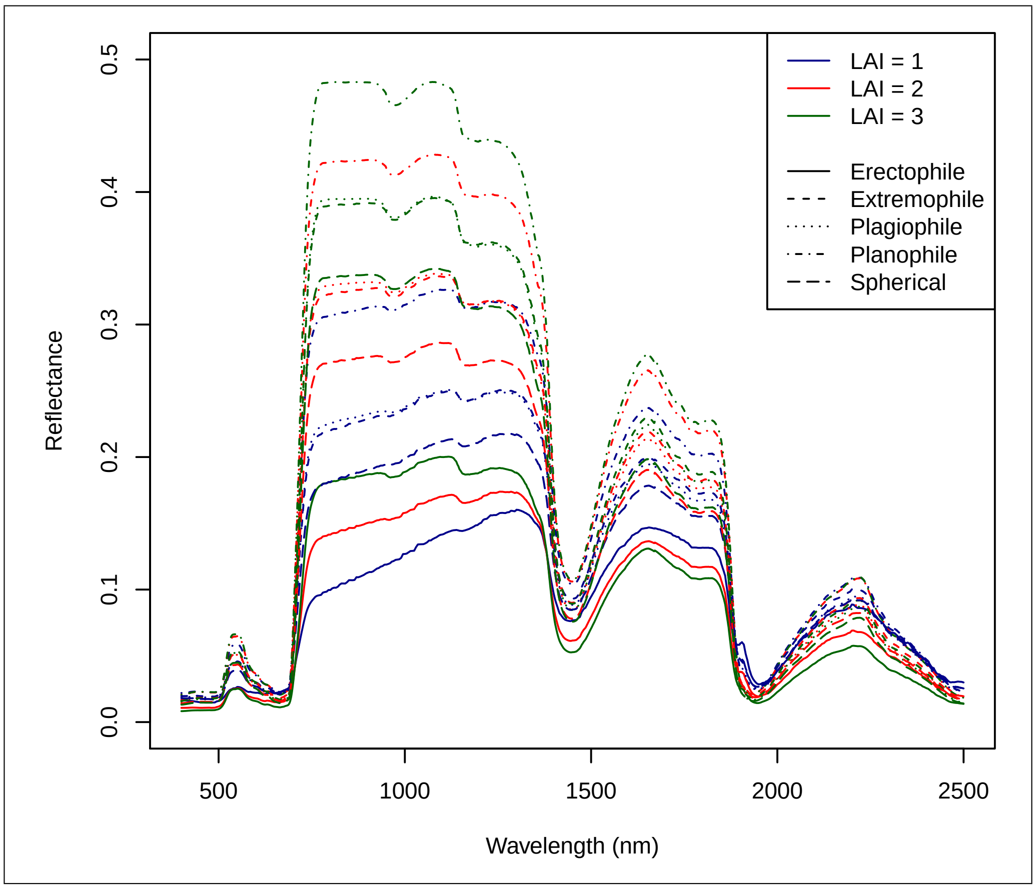
```
> ## Defining parameter
> parameter <- data.frame(LAI = rep.int(c(1,2,3),5),
+                         TypeLidf = 1,
+                         lidfa = c(rep.int(1,3), rep.int(-1,3),
+                                   rep.int(0,6), rep.int(-0.35,3)),
+                         lidfb = c(rep.int(0,6), rep.int(-1,3),
+                                   rep.int(1,3), rep.int(-0.15,3)))
> parameter

   LAI TypeLidf lidfa lidfb
1    1        1  1.00  0.00
2    2        1  1.00  0.00
3    3        1  1.00  0.00
4    1        1 -1.00  0.00
```

```
5    2        1 -1.00  0.00
6    3        1 -1.00  0.00
7    1        1  0.00 -1.00
8    2        1  0.00 -1.00
9    3        1  0.00 -1.00
10   1        1  0.00  1.00
11   2        1  0.00  1.00
12   3        1  0.00  1.00
13   1        1 -0.35 -0.15
14   2        1 -0.35 -0.15
15   3        1 -0.35 -0.15
```

```
> ## Perform simulation
> spectra <- PROSAIL(parameterList = parameter)
> spectra
```

```
Summary of Speclib


Summary of spectra
---------------------
Total number of spectra : 15
Number of bands : 2101
Width of bands : 1
```

```
> ## Plot result:
> ## Colour indicates LAI
> ## Line style indicates LIDF type
> colours <- c("darkblue", "red", "darkgreen")
> LIDF_type <- as.factor(c(rep.int("Planophile", 3),
+                          rep.int("Erectophile", 3),
+                          rep.int("Plagiophile", 3),
+                          rep.int("Extremophile", 3),
+                          rep.int("Spherical", 3)))
> plot(spectra, FUN = 1, ylim = c(0, 0.5),
+      col = colours[spectra$attributes$LAI[1]],
+      lty = which(levels(LIDF_type) == LIDF_type[1]))
> for (i in 2:nspectra(spectra))
+   plot(spectra, FUN = i, new= FALSE,
+        col = colours[spectra$attributes$LAI[i]],
+        lty = which(levels(LIDF_type) == LIDF_type[i]))
> legend("topright",
+        legend = c(paste("LAI =", c(1:3)), "", levels(LIDF_type)),
+        col = c(colours,
+                rep.int("black", 1 + length(levels(LIDF_type)))),
+        lty = c(rep.int(1, 3), 0, 1:length(levels(LIDF_type))))
```



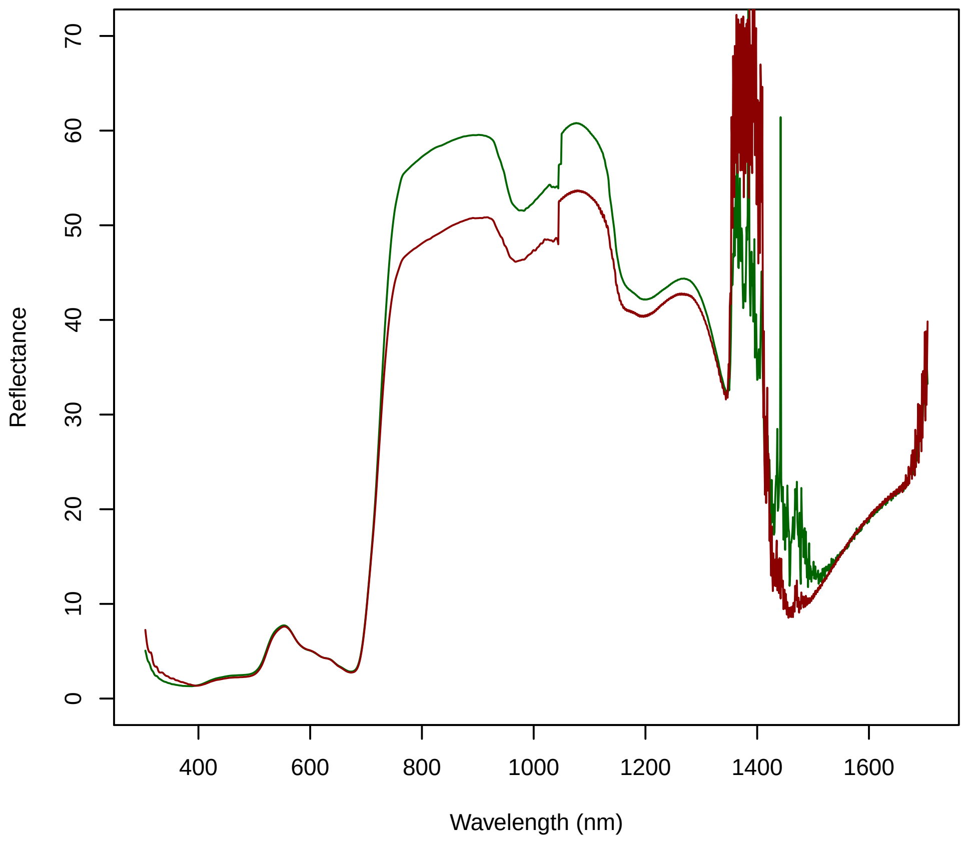
5 Basic data manipulation tools

The Speclib-class provides several routines for data manipulation which will be described in the following.

5.1 Subsets of spectra

Subsets of Speclibs can be built using the subset function. This function separates the spectra according to a condition. Usually the conditions are derived from attributes stored in the Speclib like study site, season or vegetation type. For example you could split "spectral_data" to get a subset for the summer and the spring spectra:

```
> ## Return names of attributes data
> names(attribute(spectral_data))

[1] "year"          "season"        "site"          "chlorophyll"

> ## Devide into both seasons using to the attribute "season"
> sp_spring <- subset(spectral_data, season == "spring")
> sp_summer <- subset(spectral_data, season == "summer")
> #
> #Plot results:
> #
> plot(sp_spring, FUN = "mean", col = "darkgreen", ylim = c(0,70))
> plot(sp_summer, FUN = "mean", col = "darkred", new = FALSE)
```

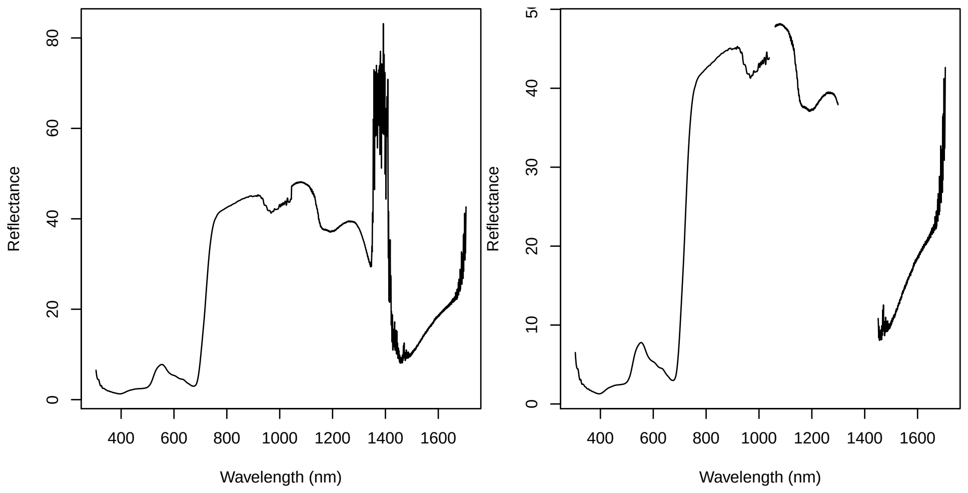


As you can see the Speclib is split into two Speclibs, one containing all spectra which had had been sampled in spring and one containing all spectra which had been acquired in summer.

5.2 Mask

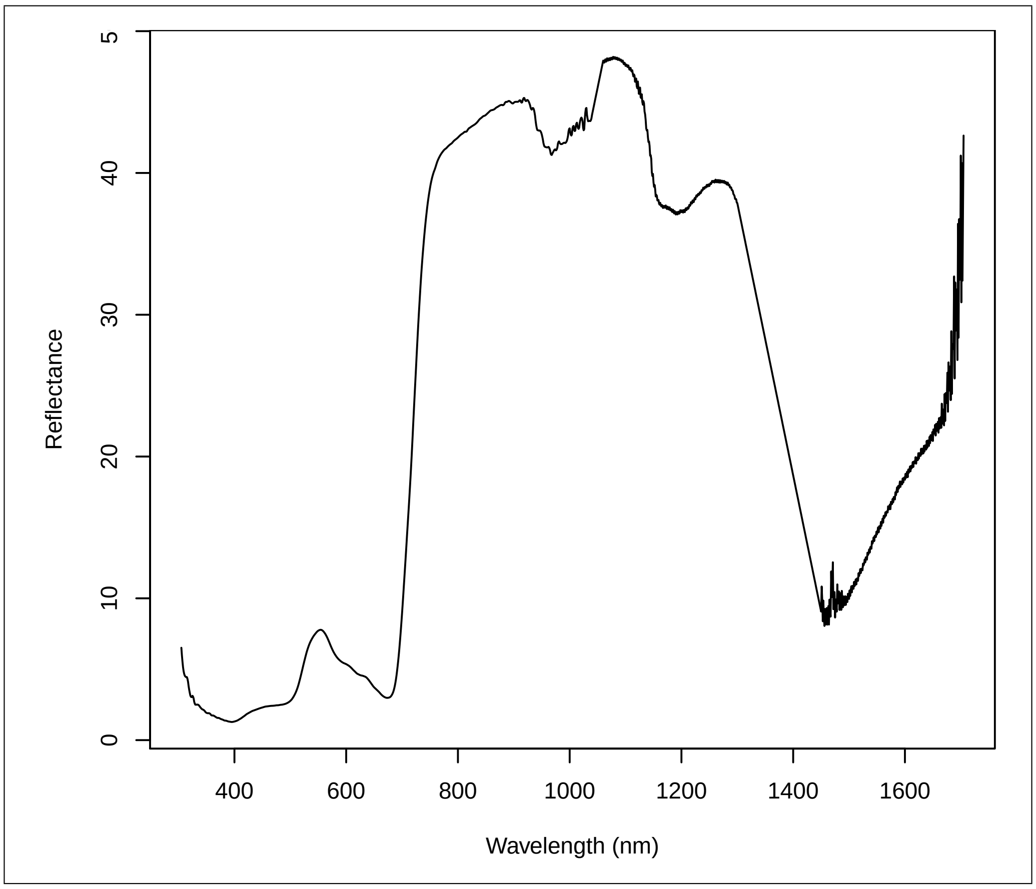
Usually, there are parts in the spectrum which are associated with errors or which are simply not of interest. hsdar allows you to mask these parts so that they don't appear in further analysis any more. In "spectral_data", the areas between 1040 and 1060 nm are errors due to channel crossing of the spectrometer and the wavelengths 1300 to 1450 are affected by water absorption. These areas should be masked in the following. There are several ways of how to enter the lower and upper limits of the wavelengths to be masked. For example you can set these values from a vector which simply consists of a sequence of lower and upper wavelengths. All wavelength between lower and upper wavelength are then masked. See ?mask for further options of how to specify these values.

```
> spectral_data_masked <- spectral_data
> mask(spectral_data_masked) <- c(1040,1060,1300,1450)
> #
> #plot results:
> #
> par(mfrow = c(1,2))
> plot(spectral_data, FUN = 1)
> plot(spectral_data_masked, FUN = 1)
```



Beside of masking these wavelength you can also assign ”new” values to them by linear interpolation. Note that interpolation is not working if start or end point of the whole spectrum were masked.

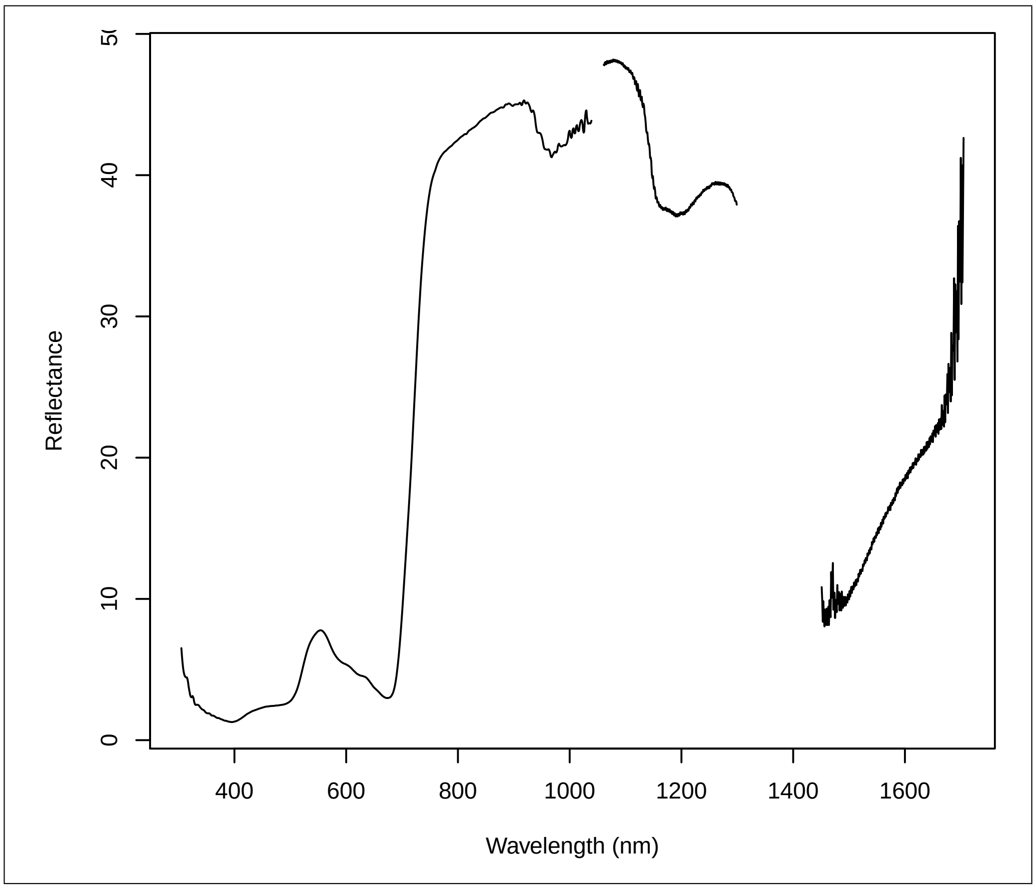
```
> spectral_data_interpolated <- interpolate.mask(spectral_data_masked)
> plot(spectral_data_interpolated, FUN = 1)
```



5.3 Filter

Having a look at one spectrum in detail you might wish to smooth the spectrum somehow.

```
> plot(spectral_data, FUN = 1, subset = c(1200,1300)) #raw spectrum
```



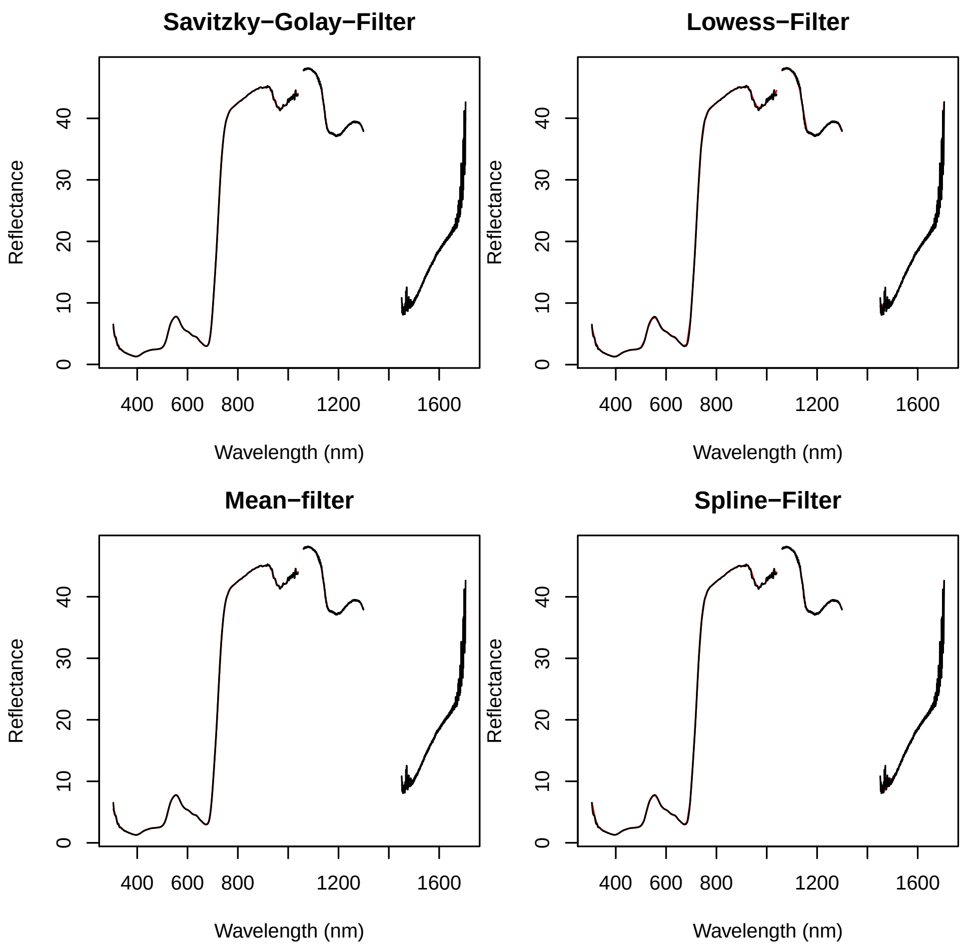
hsdar implements several methods to smooth spectra. These are Savitzky-Golay, Spline, locally weighted scatterplot smoothing (Lowess) and Mean-filter. smoothSpeclib needs a Speclib as input and the method to be used. Depending on the method there are further parameters to be set. Have a look on the hsdar help to find more information on these additional parameters.

```
> #
> #Filter Speclib:
> #
> sgolay <- smoothSpeclib(spectral_data, method = "sgolay", n = 25)
> lowess <- smoothSpeclib(spectral_data, method = "lowess", f = .01)
> meanflt <- smoothSpeclib(spectral_data, method = "mean", p = 5)
> spline <- smoothSpeclib(spectral_data, method = "spline",
+                              n = round(nbands(spectral_data)/10, 0))
```

Now plot the results:

```
> par(mfrow = c(2,2))
> plot(sgolay, FUN = 1, subset = c(1200,1300), col = "red",
+      main = "Savitzky-Golay-Filter")
```

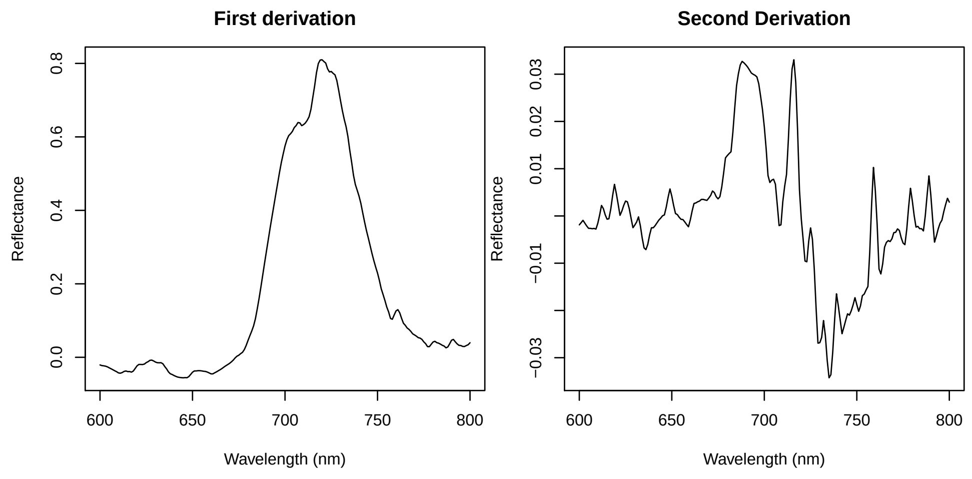
```
> plot(spectral_data, FUN = 1, new = FALSE) #raw spectrum
> plot(lowess, FUN = 1, subset = c(1200,1300), col = "red",
+      main = "Lowess-Filter")
> plot(spectral_data, FUN = 1, new = FALSE) #raw spectrum
> plot(meanflt, FUN = 1, subset = c(1200,1300), col = "red",
+      main = "Mean-filter")
> plot(spectral_data, FUN = 1, new = FALSE) #raw spectrum
> plot(spline, FUN = 1, subset = c(1200,1300), col = "red",
+      main = "Spline-Filter")
> plot(spectral_data, FUN = 1, new = FALSE) #raw spectrum
```



5.4 Calculations of derivations

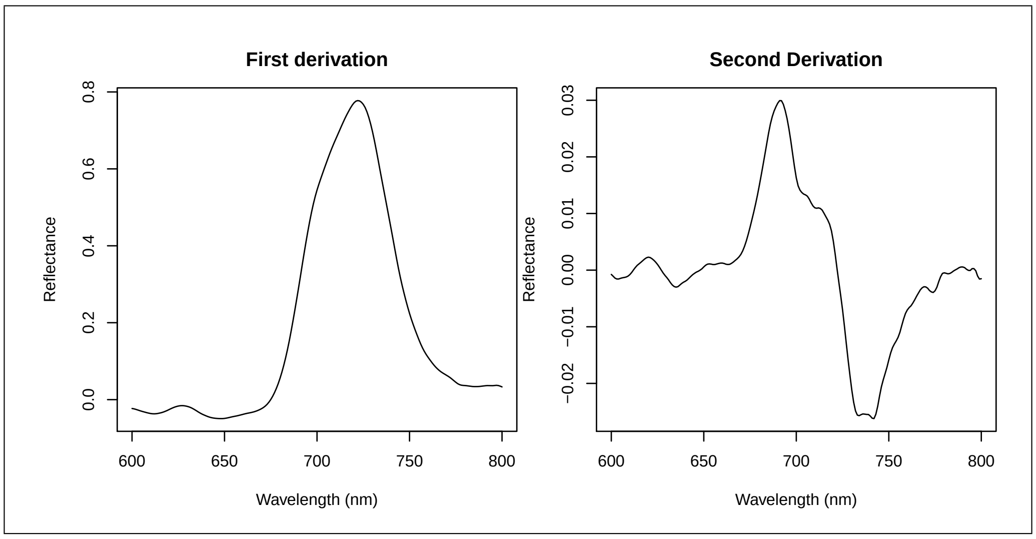
The derivation of spectra are needed in some analyzing techniques for example to characterize the shape of the red edge. The number of derivation is indicated by the parameter m, thus m = 1 returns the first derivation of the spectra.

```
> spectral_data_1deriv <- derivative.speclib(spectral_data, m = 1)
> spectral_data_2deriv <- derivative.speclib(spectral_data, m = 2)
> ## Get index of red edge wavelength
> redEdgePart <- wavelength(spectral_data_2deriv) >= 600 &
+                wavelength(spectral_data_2deriv) <= 800
> ## Cut spectra to red edge
> spectral_data_1deriv <- spectral_data_1deriv[,redEdgePart]
> spectral_data_2deriv <- spectral_data_2deriv[,redEdgePart]
> #
> #plot derivations of the red edge area of 1. spectrum in the Speclib:
> #
> par(mfrow=c(1,2))
> plot(spectral_data_1deriv, FUN = 1, xlim = c(600,800),
+      main = "First derivation")
> plot(spectral_data_2deriv, FUN = 1, xlim = c(600,800),
+      main = "Second Derivation")
```



However, depending on the objectives it might be favorable to first smooth the spectra before calculating the derivations:

```
> spectral_data_1deriv <- derivative.speclib(smoothSpeclib(
+   spectral_data, method = "sgolay", n = 35), m = 1)
> spectral_data_2deriv <- derivative.speclib(smoothSpeclib(
+   spectral_data, method = "sgolay", n = 35), m = 2)
> #
> #Plot results:
> #
> par(mfrow=c(1,2))
> plot(spectral_data_1deriv, FUN = 1, xlim = c(600,800),
+      main = "First derivation")
> plot(spectral_data_2deriv, FUN = 1, xlim = c(600,800),
+      main = "Second Derivation")
```



5.5 Resampling of bands to various satellite sensors

hsdar allows to resample the speclib to the bands of common satellite sensors. The characteristics of (satellite) sensor to integrate spectra can be chosen from a list of already implemented sensors or they can be passed as a data.frame with two columns: first column with lower bounds of channels and second column with upper bounds. See which sensors are already implemented:

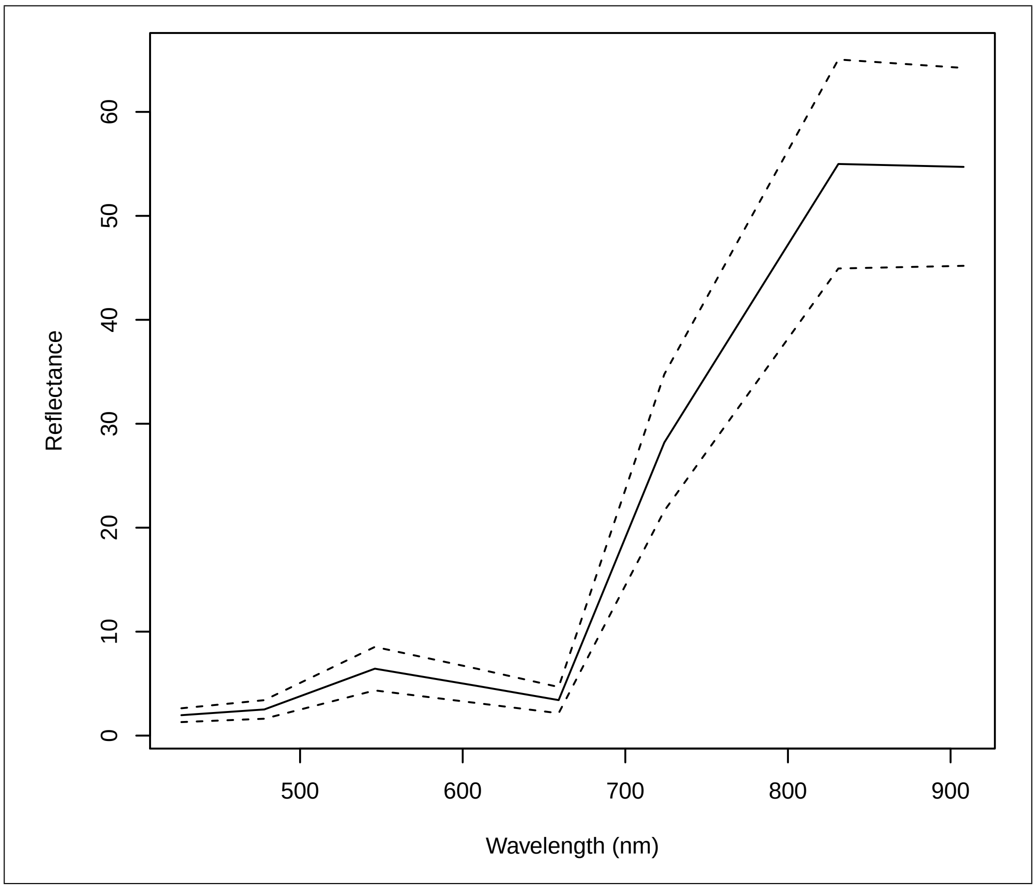
```
> get.sensor.characteristics(0)

             Numerical abbreviation Response function implemented
ALI                               5                         FALSE
EnMAP                            11                         FALSE
Hyperion                          6                         FALSE
Landsat4                          9                          TRUE
Landsat5                          4                          TRUE
Landsat7                         10                          TRUE
Landsat8                         12                          TRUE
MODIS                             1                         FALSE
Quickbird                         7                          TRUE
RapidEye                          2                          TRUE
Sentinel2                        13                          TRUE
WorldView2-4                      8                          TRUE
WorldView2-8                      3                          TRUE
```

For some sensors the spectral response functions are available. The spectra can be resampled in three ways. One possibility is the use of the spectral response functions, if available. Otherwise spectra can be resampled assuming a Gaussian distribution (responsefunction = FALSE) or the mean value (responsefunction = NA) of reflectances between the limiting wavelength of a satellites channel.

```
> ## use spectral response function
> spectral_data_resampled <- spectralResampling(spectral_data,
+                                               "WorldView2-8")
```

See what changed in the Speclib:

```
> spectral_data_resampled

Summary of Speclib


History of usage
---------------------
(1)   Reflectance = mean applied to matrix spectra by attribute 'site'
(2)   Apply mask to spectra
(3)   Integrated spectra to WorldView2-8 channels


Summary of spectra
---------------------
Total number of spectra : 45
Number of bands : 8
Mean width of bands : 66.25

> wavelength(spectral_data_resampled)

[1] 427 478 546 608 659 724 831 908

> #
> #plot results:
> plot(spectral_data_resampled)
```



You can see that the number of bands is now reduced to the 8 WorldView2 bands instead of 1401 hyperspectral channels. The wavelengths are automatically adapted in the speclib attribute. Note that standard deviations are now not longer plotted as dashed line by default but as error bars because the data are not longer continuous.

6 Continuum removal

Continuum removal is a commonly used method in hyperspectral remote sensing to normalize spectra and to detect and ensure the comparability of absorption features The continuum removal transformation is performed by firstly establishing a continuum line/hull which connects the local maxima of the reflectance spectrum. Two kinds of this hull are well established in scientific community: the convex hull (e.g. Mutanga and Skidmore (2004a)) and the segmented hull (e.g. Clark et al. (1987)). Both hulls are established by connecting the local maxima, however, the precondition of the convex hull is that the resulting continuum line must be convex whereas considering the segmented hull it might be concave or convex but the algebraic sign of the slope is not allowed to change from the global maximum of the spectrum downwards to the sides. In contrast to a convex hull, the segmented hull is able to identify small absorption features.

Because the continuum removal transformation is sensitive to errors in the spectrum, it's advisable to first mask erroneous parts of the spectrum. Further you should consider to smooth the spectra very slightly to avoid small local maxima which are not associated with reflection maxima. Read section 5.3 and 5.2 for more details on the preprocessing.

```
> data(spectral_data)
> spectral_data_preproc <- smoothSpeclib(spectral_data,
+                                       method = "sgolay", n = 5)
> mask(spectral_data_preproc) <- c(1040,1060,1300,1450)
```

Then have a look at the transformSpeclib function:

```
> str(transformSpeclib)

function (data, ..., method = "ch", out = "bd")
```

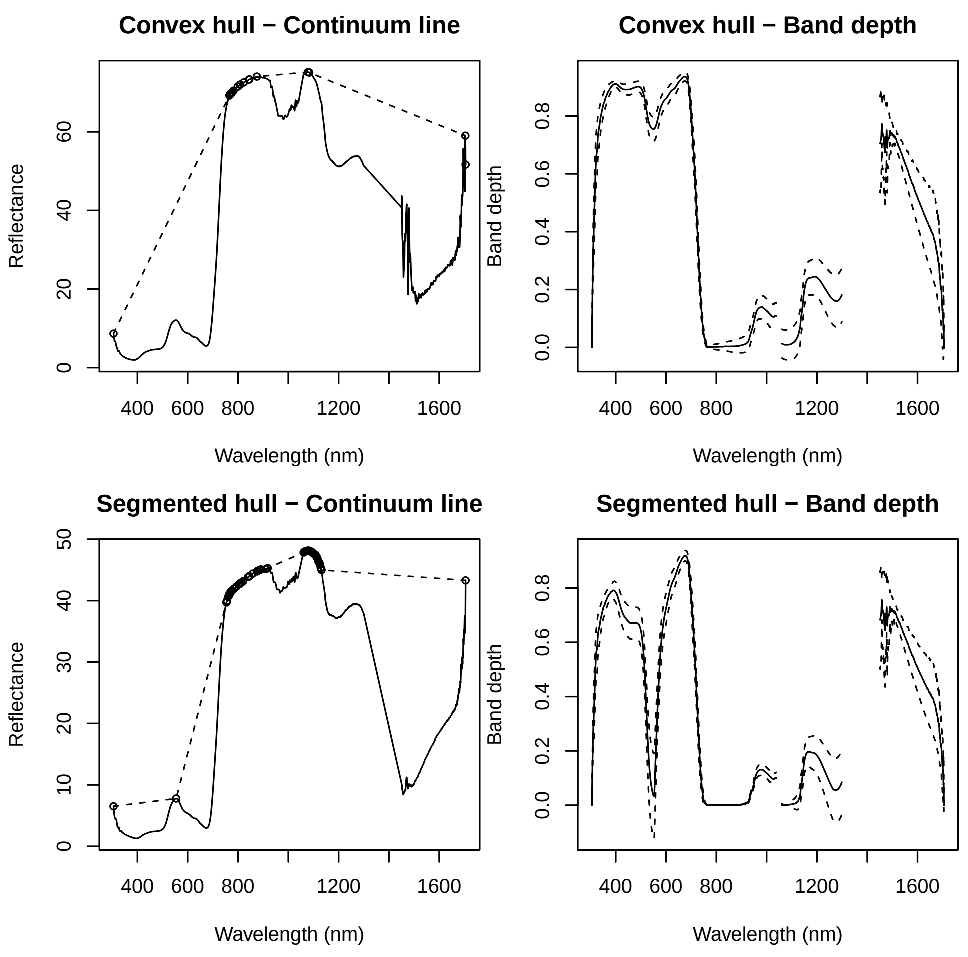
”data” defines the speclib which is to be transformed. Concerning the ”methods” parameter, currently the mentioned **c**onvex **h**ull (”ch”) and the **s**egmented **h**ull (”sh”) are implemented. The ”out” parameter indicates if the continuum line (”raw”), the continuum removed spectra (**b**and **d**epth, ”bd”) or the ”ratio” will be returned. Have a look on the help page of transform.speclib and the listed literature for details on these methods or for help with interpretation. The output type of ”ratio” the ”bd” is a speclib, out=”raw” returns an object of ”clman”. ”clman” is a class designed to store and handle manual continuum lines.

The following example will show you how to calculate the continuum line (just for visualization) and the band depth using the convex hull as well as the segmented hull approach:

```
> #convex hull:
> ch_cline <- transformSpeclib(spectral_data_preproc[16:30,],
+                              method = "ch", out = "raw")
> ch_bd <- transformSpeclib(spectral_data_preproc,
+                           method = "ch", out = "bd")
> #
> #segmented hull:
> #
> sh_cline <- transformSpeclib(spectral_data_preproc,
+                              method = "sh", out = "raw")
> sh_bd <- transformSpeclib(spectral_data_preproc,
+                           method = "sh", out = "bd")
```

Plot continuum lines and resulting band depths for both methods to see the differences:

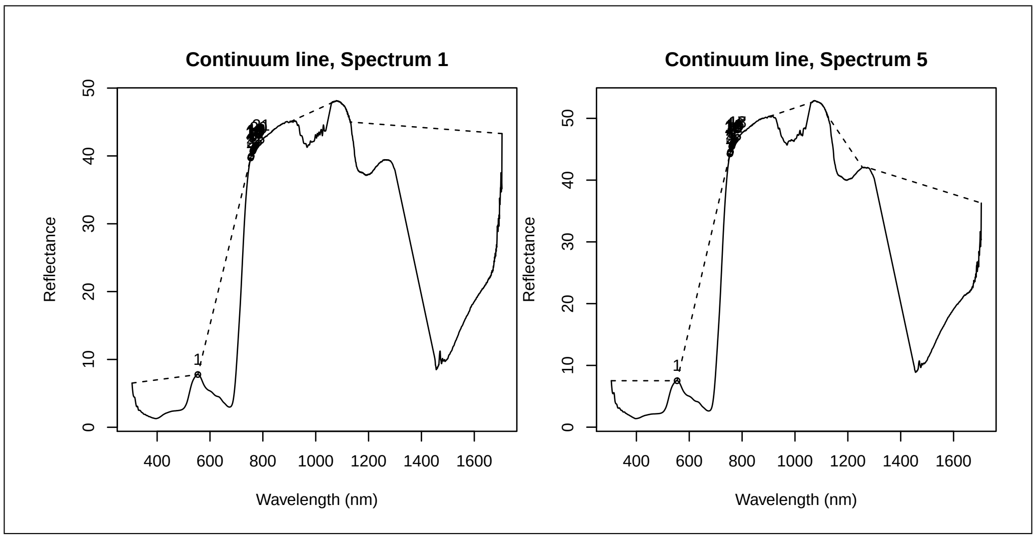
```
> #plot results for the first spectrum:
> #
> par(mfrow = c(2,2))
> plot(ch_cline, ispec = 1, numeratepoints = FALSE,
+      main = "Convex hull - Continuum line")
> plot(ch_bd, ispec = 1, main = "Convex hull - Band depth")
> plot(sh_cline, ispec = 1, numeratepoints = FALSE,
+      main = "Segmented hull - Continuum line")
> plot(sh_bd, ispec = 1, main = "Segmented hull - Band depth")
```



6.1 Manually adapting continuum lines

Let's have a look on the segmented hull in more detail and compare it to another spectrum. Therefore, we zoom to the red edge area:

```
> par(mfrow = c(1,2))
> plot(sh_cline, ispec = 1, main = "Continuum line, Spectrum 1",
+      subset = c(500,800)) #first spectrum
> plot(sh_cline, ispec = 5, main = "Continuum line, Spectrum 5",
+      subset = c(500,800)) #fifth spectrum
```



By the way: Plotting an object of the class "clman" allows you to numerate the local maxima which were used to construct this line.

Comparing the first spectrum and the fifth spectrum it is obvious that in the first there are several small local maxima around 600 nm whilst the fifth spectrum features clearly one larger absorption feature between the local maxima around 550 nm and 750 nm. Thus, if your objectives include to compare the absorption in the red edge of different spectra, these two spectra would not be comparable due to the fact that this large feature is split into several smaller features in spectrum 1.

However, you might have the impression that some of the local maxima could be removed because they are very small and maybe afflicted with uncertainties which might legitimate it to manipulate the continuum line. Therefore, hsdar provides functions to remove and add "continuum points" to a continuum line which allows to adapt the continuum line which can be used to also adapt band depth or ratio transformation. Handle these functions with care to avoid continuum lines too much build by subjective methods.

If you have a large Speclib, its quite labor-intensive to manually adapt the continuum lines because you have to go through every sample in you Speclib. In the following example the procedure will be shown with one exemplary sample:

Continuum points can be deleted using the deletecp function:

```
> str(deletecp)

function (x, ispec, cpdelete)
```

with x is the continuum line, ispec is the name or index of the spectrum to be modified and cpdelete is a single value or vector of wavelength containing fix points to be deleted. Comparing spectrum 1 and spectrum 5 we have seen that the continuum line of spectrum 1 features additional local maxima beyond 580 nm which are, however, not physically explainable so that you might want to delete all points between 580nm and 700nm:

```
> getcp(sh_cline, 1, subset = c(500, 700)) #see all points
```

```
$ptscon
  Wavelength Reflectance
1        554    7.779095

$ispec
[1] 1

> sh_cline <- deletecp(sh_cline, 1,
+                       c(500:700)) #delete all between 500 and 700 nm
> getcp(sh_cline, 1, subset = c(500, 700)) #see what happened

$ptscon
[1] Wavelength  Reflectance
<0 rows> (or 0-length row.names)

$ispec
[1] 1
```

Similarly you could add a continuum point by specifying the wavelength of the point to be added. Though it doesn't make sense in this context you could add a point at the wavelength 460:

```
> #sh_cline <- addcp(sh_cline, 1, 460)
```

After modifying the continuum line by adding and/or deleting continuum points, you can check the line for intersection with the spectrum using the checkhull function:

```
> checkhull(sh_cline, 1)$error

[1] 1060 1060
```

If there are any errors, additional continuum points have to be defined to meet the constraint that the hull does not cross the spectrum. In this case, we have to add a point at 1060 nm which is an issue due to the mask. Please note that the point at 1060 nm does not affect the following example.

```
> sh_cline <- addcp(sh_cline, 1, 1060)
```

Again, we check for errors:

```
> checkhull(sh_cline, 1)$error

[1] 0 0
```

After all uncertainties are removed, the hull can be re-calculated using ”makehull”:

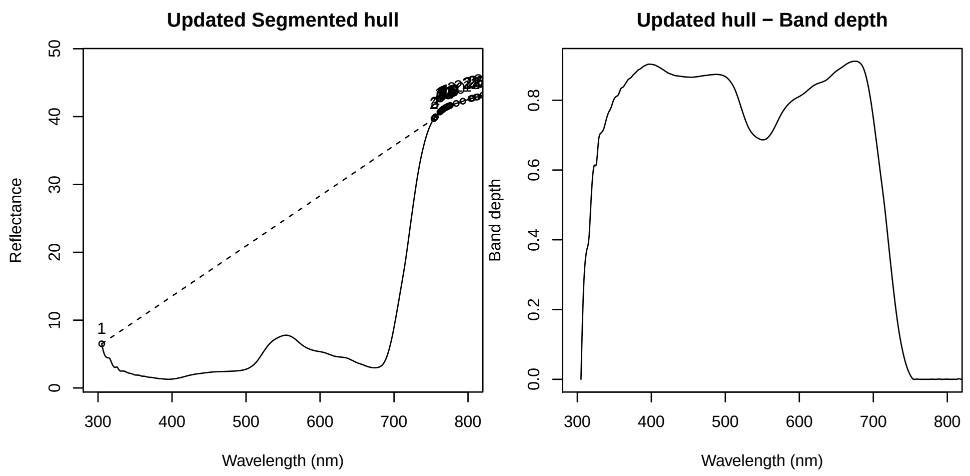
```
> sh_clineUpdate <- makehull(sh_cline, 1) #update the hull of spectrum 1
```

After all hulls of the Speclib are modified and corrected, the transformed Speclib has to be updated with the new hulls:

```
> sh_bd <- updatecl(sh_bd, sh_clineUpdate) #update the band depth
```

Now, we can plot the resulting continuum removed spectra between 300 and 800 nm:

```
> #plot new line:
> par (mfrow = c(1,2))
> plot(sh_cline, ispec = 1, main = "Updated Segmented hull",
+      xlim = c(300,800))
> #plot new band depth
> plot(sh_bd[1,], main="Updated hull - Band depth",
+      xlim = c(300,800))
```

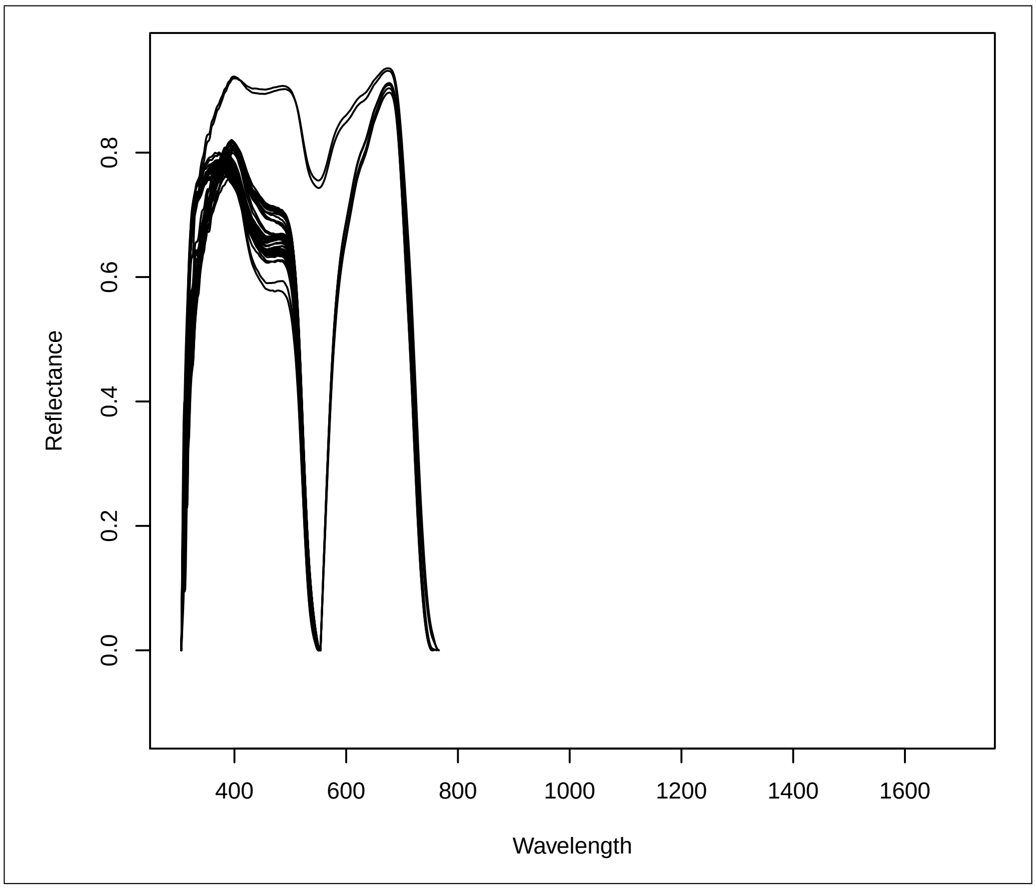


```
> data(spectral_data)
> spectral_data_preproc <- smoothSpeclib(spectral_data,
+                                        method = "sgolay", n = 5)
> mask(spectral_data_preproc) <- c(1040,1060,1300,1450)
> sh_cline <- transformSpeclib(spectral_data_preproc,
+                              method = "sh", out = "raw")
> par (mfrow = c(2,1))
> plot(sh_cline, 1, subset = c(550, 650))
> plot(sh_cline, 5, subset = c(550, 650))
> getcp(sh_cline, 1, subset = c(500, 700))
> getcp(sh_cline, 5, subset = c(500, 700))
> sh_cline <- deletecp(sh_cline, 1, c(550:700)) #delete
> checkhull(sh_cline, 1)$error
> sh_cline <- addcp(sh_cline, 1, 560)
> sh_clineUpdate <- makehull(sh_cline, 1)
> sh_bd <- transformSpeclib(spectral_data_preproc,
+                           method = "sh", out = "bd")
> sh_bd <- updatecl(sh_bd, sh_clineUpdate)
> par (mfrow = c(1,2))
> plot(sh_cline, 1, main = "Updated Segmented hull",
+      subset = c(500,800))
> #plot new band depth
> plot(sh_bd, 1, main="Updated hull - Band depth",
+      subset = c(500,800))
>
```

6.2 Extracting absorption features

Let's continue with the preprocessed spectra from section 6

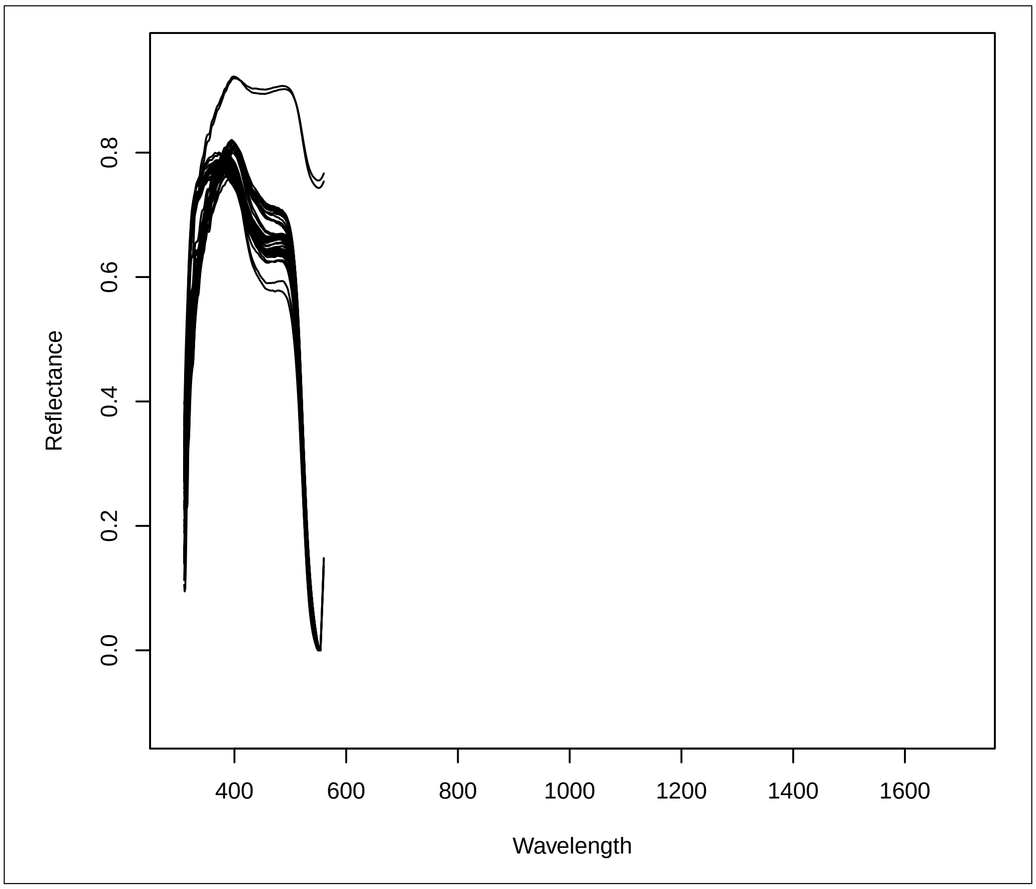
```
> sh_bd <- transformSpeclib(spectral_data_preproc,
+                           method = "sh", out = "bd")
> ## Define features automatically
> features <- define.features(sh_bd)
> ##Example to isolate the features around 450,700,1200 and 1500nm.
> featureSelection <- specfeat(features, c(450,700,1200,1500))
> ## Plot features
> plot(featureSelection, 1:4)
```



Some features are larger than others. For some research questions it might be important to cut the features at specific wavelengths.

The following example shows how to cut the first two features:

```
> featuresCut <- cut_specfeat(featureSelection, fnumber = c(1,2),
+                             limits = c(c(310, 560), c(589, 800)))
> ## Plot result
> plot(featuresCut, 1:2)
```



7 Basic data analysis tools

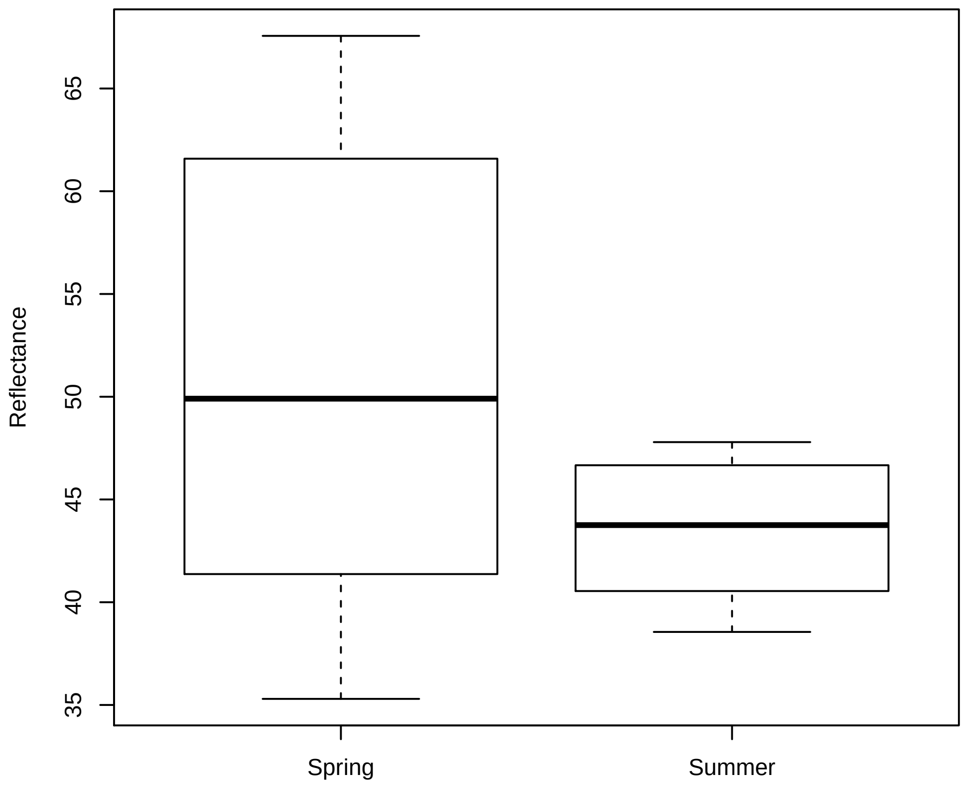
7.1 T-Test

You can compare the distribution of two subsets of a Speclib by a T-test. At each wavelength subset one is compared to subset two: First, split "spectral_data" into Speclibs of summer and spring spectra:

```
> #split into subsets:
> sp_spring <- subset(spectral_data, season == "spring")
> sp_summer <- subset(spectral_data, season == "summer")
```

Then perform a T-test: The table shows the statistical parameters of the T-test (e.g. p-value, confidence intervals, mean values) for each wavelength. E.g. for the wavelength 750 nm the differences are highly significant with higher reflectance at the Kailash site (31.11%) than on the Namco site (24.99%). Let's visualize how the reflexion at the red edge reflexion shoulder (around 750 nm) of the Kailash samples compares to the reflexion at the same position of the Namco sites:

```
> boxplot(spectra(sp_spring)[,wavelength(sp_spring) == 750],
+         spectra(sp_summer)[,wavelength(sp_summer) == 750],
+         names=c("Spring","Summer"), ylab = "Reflectance")
```



7.2 Regressions

Currently, hsdar provides no special tools for regressions, however you can use the standard R routines. E.g. if you want to test each wavelength for its relation to plant fraction, write a loop over each wavelength and access the spectra and the attribute ”chlorophyll” from the Speclib:

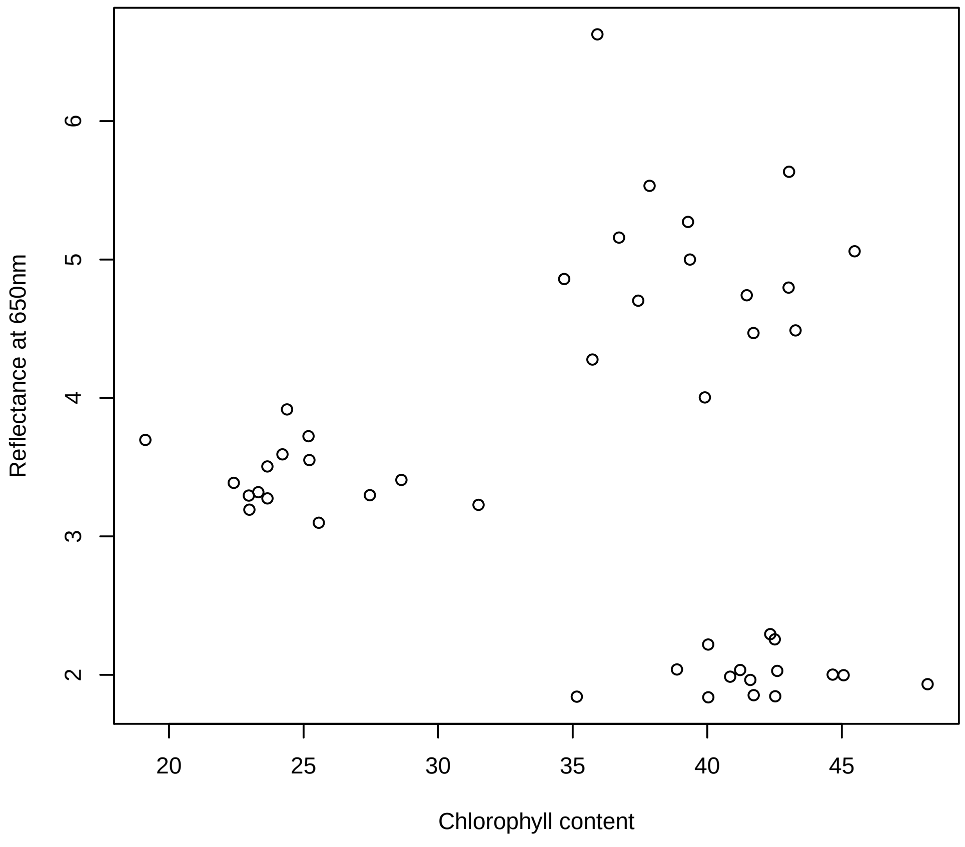
```
> result <- list()
> for (i in 1:length(wavelength(spectral_data)))
+ {
+   result[[i]] <- summary(lm(spectra(spectral_data)[,i] ~
+                              attribute(spectral_data)$chlorophyll))
+ }
> names(result) <- wavelength(spectral_data)
```

In this way you can access the results using the wavelengths as string. E.g. let’s have a look on the results at 650 nm:

```
> result$"650"

Call:
lm(formula = spectra(spectral_data)[, i] ~ attribute(spectral_data)$chlorophyll)

Residuals:
   Min     1Q Median     3Q    Max
-1.633 -1.327 -0.284  1.137  3.163

Coefficients:
                                     Estimate Std. Error t value
(Intercept)                           4.01383    0.85816   4.677
attribute(spectral_data)$chlorophyll -0.01531    0.02363  -0.648
                                     Pr(>|t|)
(Intercept)                          2.89e-05 ***
attribute(spectral_data)$chlorophyll    0.521
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.294 on 43 degrees of freedom
Multiple R-squared:  0.009667,        Adjusted R-squared:  -0.01336
F-statistic: 0.4198 on 1 and 43 DF,  p-value: 0.5205
```

You can easily plot this relation by a scatterplot with PV on th the x-axis and the reflectance at the wavelength 650 nm on the y-axis.

```
> plot(attribute(spectral_data)$chlorophyll,
+      spectra(spectral_data)[,wavelength(spectral_data)==650],
+      xlab = "Chlorophyll content", ylab = "Reflectance at 650nm")
```



8 Calculating spectral indices

There are three different kinds of spectral indices implemented in hsdar: A variety of common as well as recently developed vegetation indices, red edge parameters and normalised ratio indices.

8.1 Vegetation indices

To see the whole set of implemented indices and how they are calculated please read the hsdar help or manual. The indices are calculated for each sample in the Speclib. For example calculate the NDVI like this:

```
> data(spectral_data)
> ndvi <- vegindex(spectral_data, "NDVI")
> ndvi #see ndvi

 [1] 0.8645385 0.8902434 0.9030130 0.8645695 0.8912563 0.8744318
 [7] 0.8859704 0.8970170 0.8914286 0.8804466 0.8873673 0.8921778
[13] 0.9007679 0.8898776 0.8911217 0.8518171 0.8957341 0.8821210
[19] 0.8821841 0.8839156 0.8798848 0.8740733 0.8639535 0.8770984
[25] 0.8866471 0.8864226 0.8872358 0.8845266 0.8778616 0.9002123
[31] 0.9190947 0.9293130 0.9151203 0.9307388 0.9335307 0.9178542
[37] 0.9393256 0.9354712 0.9271099 0.9375971 0.9321078 0.9296322
[43] 0.9350494 0.9262040 0.9327303
```

You can also directly calculate all available indices by creating a vector of the names of all already implemented indices with "vegindex()" which is used as index parameter in the vegindex function:

```
> avl <- vegindex()
> vi <- vegindex(spectral_data, index = avl)
```

8.2 Red edge parameters

Shape and location of the red edge are commonly described by four parameters:

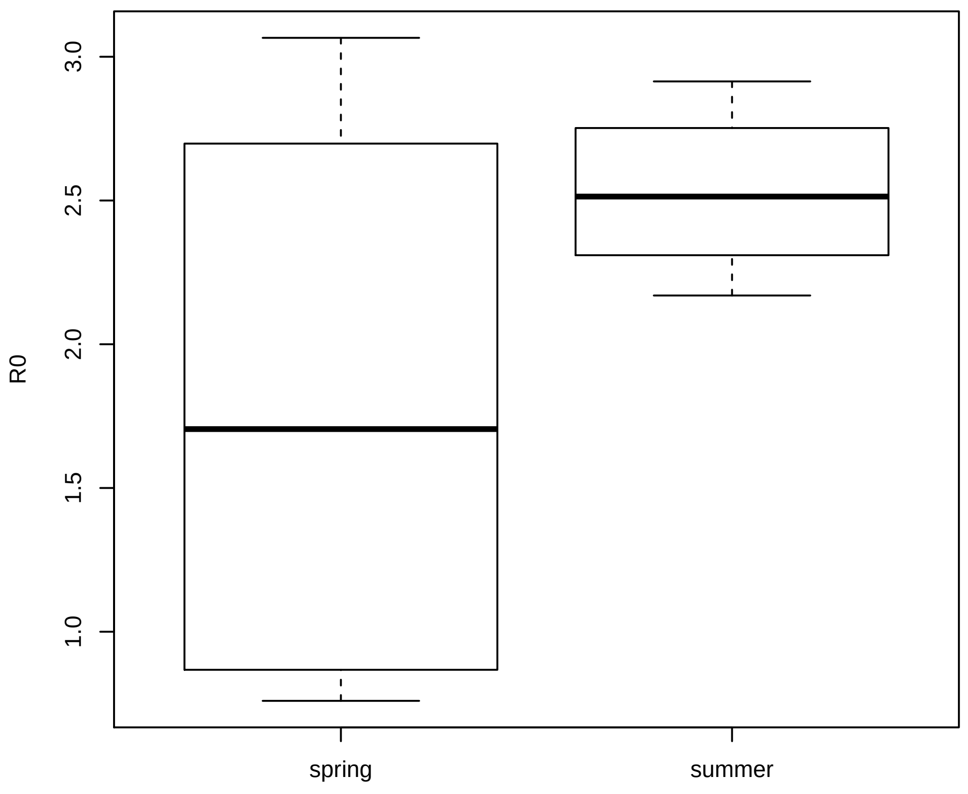
R0	the minimum reflectance in the red spectrum
l0	wavelength of the minimum reflectance
lp	inflection point
Rs	shoulder wavelength

The red edge parameters are calculated as proposed in Bach (1995) from the spectral area between 600 and 900 nm. l0 is calculated as the last root before the maximum value of the 2nd derivation. The minimum reflectance is the reflectance at (l0). The inflection point is the root of the 2nd derivative function between the maximum value and the minimum value. The shoulder wavelength is the first root beyond the minimum value of the 2nd derivation.

```
> data(spectral_data)
> rd <- rededge(spectral_data)
```

Results can be presented as boxplot. For example create a boxplot for R0:

```
> boxplot(rd$R0 ~ spectral_data$attributes$season, ylab = "R0")
```



8.3 Normalised ratio indices

hsdar has implemented a method to calculate NDVI-like Normalised ratio indices (NRI) (also named as narrow band indices). Thus for all possible band combinations in the spectrum, the following calculation is performed:

$$nri_{B1,B2} = \frac{R_{B1} - R_{B2}}{R_{B1} + R_{B2}} \tag{1}$$

with R being reflectance values at wavelength $B1$ and $B2$, respectively.

With this function you could now calculate the NRI for all band combinations, however this requires some time so that we will explain the NRI using resampled bands. Resample ”spectral_data to the resolution of WorldView-2-8:

```
> spec_WV <- spectralResampling(spectral_data, "WorldView2-8",
+                                response_function = FALSE)
```

Now, see how nris are calculated:

```
> str(nri)

function (x, b1, b2, recursive = FALSE, bywavelength = TRUE)

> help(nri)
```

There are two possibilities what to do with nri. Either you could assign two bands by wavelength from which the NRI should be calculated or you can assign "recursive=TRUE" which means that NRI are calculated for all possible band combinations. For our case with 8 WorldView channels this would mean that 8*7 = 56 combinations will be calculated for each spectrum in the Speclib.

```
> nri_WV <- nri(spec_WV, recursive = TRUE)
> nri_WV

Data: nri, dimension: 8, 8, 45
          [,1]      [,2]        [,3]        [,4]      [,5]      [,6]
[1,]        NA        NA          NA          NA        NA        NA
[2,] 0.2002899        NA          NA          NA        NA        NA
[3,] 0.5084916 0.3431500          NA          NA        NA        NA
[4,] 0.4606805 0.2868589 -0.06243717          NA        NA        NA
[5,] 0.4664228 0.2935569 -0.05514847 0.007313881        NA        NA
[6,] 0.8562469 0.7917383  0.61592603 0.653241754 0.6490287        NA
[7,] 0.9133002 0.8726372  0.75581145 0.781375051 0.7785102 0.2617243
[8,] 0.9174453 0.8786030  0.76656735 0.791138887 0.7883868 0.2853859
           [,7] [,8]
[1,]         NA   NA
[2,]         NA   NA
[3,]         NA   NA
[4,]         NA   NA
[5,]         NA   NA
[6,]         NA   NA
[7,]         NA   NA
[8,] 0.02557163   NA

    ... (43 layers omitted)

          [,1]      [,2]        [,3]       [,4]      [,5]      [,6]
[1,]        NA        NA          NA         NA        NA        NA
[2,] 0.1809942        NA          NA         NA        NA        NA
[3,] 0.4942798 0.3440664          NA         NA        NA        NA
```

```
[4,] 0.4062179 0.2430969 -0.11018558         NA        NA        NA
[5,] 0.4495496 0.2923421 -0.05750886 0.05301264        NA        NA
[6,] 0.8970190 0.8548073  0.72354215 0.77216750 0.7498497        NA
[7,] 0.9488000 0.9269964  0.85592633 0.88284975 0.8705822 0.3477377
[8,] 0.9509982 0.9300971  0.86182916 0.88771623 0.8759247 0.3673642
           [,7] [,8]
[1,]         NA   NA
[2,]         NA   NA
[3,]         NA   NA
[4,]         NA   NA
[5,]         NA   NA
[6,]         NA   NA
[7,]         NA   NA
[8,] 0.02250085   NA
      wavelength of length = 8
      fwhm for each wavelength


History of usage
---------------------
(1)   Reflectance = mean applied to matrix spectra by attribute 'site'
(2)   Integrated spectra to WorldView2-8 channels
(3)   NRI values calculated
```

To get access to the NRI of a specific spectrum use

```
> str(nri_WV)
```

to see that the NRI can be accessed via object$nri. This object has three dimensions: band 1, band 2 and the spectrum. Therefore type the following to see the NRI for all band combinations of the first spectrum in the Speclib:

```
> nri_WV$nri[,,1]

          [,1]      [,2]        [,3]        [,4]      [,5]      [,6]
[1,]        NA        NA          NA          NA        NA        NA
[2,] 0.2002899        NA          NA          NA        NA        NA
[3,] 0.5084916 0.3431500          NA          NA        NA        NA
[4,] 0.4606805 0.2868589 -0.06243717          NA        NA        NA
[5,] 0.4664228 0.2935569 -0.05514847 0.007313881        NA        NA
[6,] 0.8562469 0.7917383  0.61592603 0.653241754 0.6490287        NA
[7,] 0.9133002 0.8726372  0.75581145 0.781375051 0.7785102 0.2617243
[8,] 0.9174453 0.8786030  0.76656735 0.791138887 0.7883868 0.2853859
           [,7] [,8]
[1,]         NA   NA
[2,]         NA   NA
[3,]         NA   NA
```

```
[4,]         NA   NA
[5,]         NA   NA
[6,]         NA   NA
[7,]         NA   NA
[8,] 0.02557163   NA
```

Note that the resulting matrix only contains the indices for one side of the matrix because the information content would be the same for the other side of the matrix only with opposite algebraic sign.

8.4 Comparing distributions of NRI

See section 9 for how to relate NRIs to environmental variables which is most likely what you want to achieve with your NRIs.

9 Analysing relations between NRI and environmental variables

This section will show how to relate the NRI (see section 8.3) to environmental variables and how to create nice plots with a lot of information content.

If you haven't already calculated the NRI from WorldView-resampled bands, do it now to work through this section:

```
> spec_WV <- spectralResampling(spectral_data, "WorldView2-8",
+                               response_function = FALSE)
> nri_WV <- nri(spec_WV, recursive = TRUE)
```

9.1 Correlations

In this example we want to correlate each NRI to the chlorophyll content of the vegetation. Use the Speclib and NRI data created in section 9. First create a new variable from the attributes of the Speclib containing the chlorophyll content per sample:

```
> chlorophyll <- attribute(spec_WV)$chlorophyll
```

Then you can correlate this to the NRI:

```
> cortestnri <- cor.test(nri_WV, chlorophyll)
```

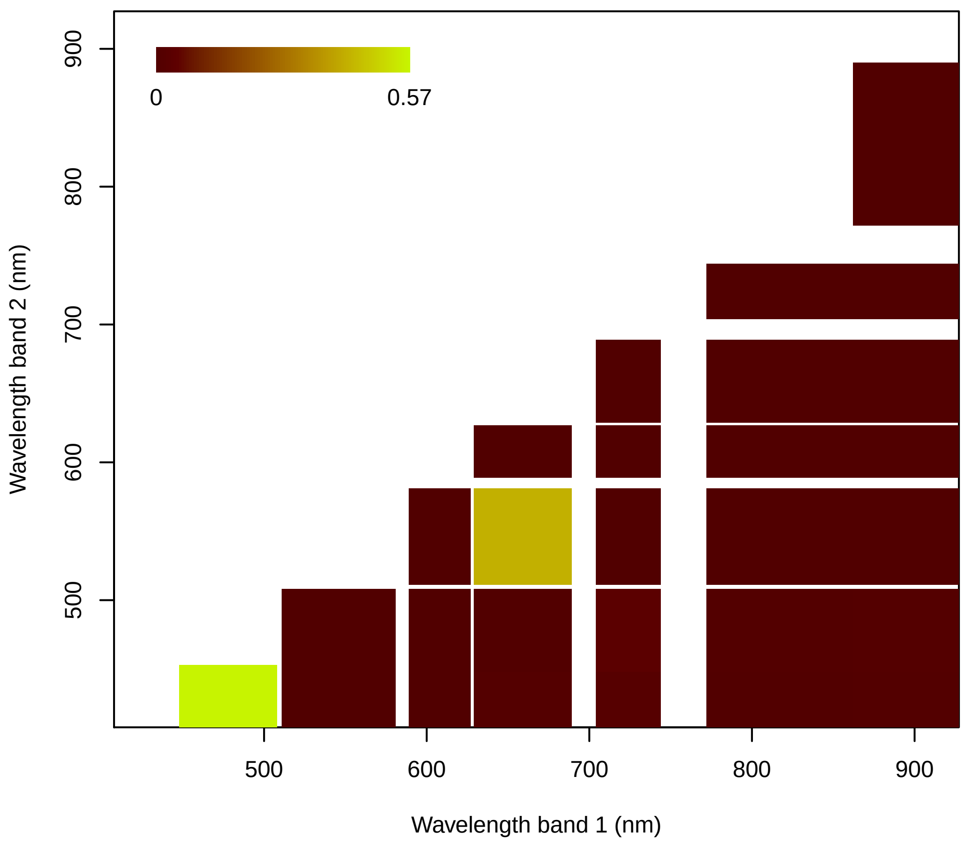
See how the output of such a correlation is printed:

```
> cortestnri

Data: nri, dimension: 8, 8, 45
          [,1]      [,2]        [,3]        [,4]      [,5]      [,6]
[1,]        NA        NA          NA          NA        NA        NA
[2,] 0.2002899        NA          NA          NA        NA        NA
[3,] 0.5084916 0.3431500          NA          NA        NA        NA
[4,] 0.4606805 0.2868589 -0.06243717          NA        NA        NA
[5,] 0.4664228 0.2935569 -0.05514847 0.007313881        NA        NA
[6,] 0.8562469 0.7917383  0.61592603 0.653241754 0.6490287        NA
[7,] 0.9133002 0.8726372  0.75581145 0.781375051 0.7785102 0.2617243
[8,] 0.9174453 0.8786030  0.76656735 0.791138887 0.7883868 0.2853859
           [,7] [,8]
[1,]         NA   NA
[2,]         NA   NA
[3,]         NA   NA
[4,]         NA   NA
[5,]         NA   NA
[6,]         NA   NA
[7,]         NA   NA
[8,] 0.02557163   NA

    ... (43 layers omitted)

          [,1]      [,2]        [,3]       [,4]      [,5]      [,6]
[1,]        NA        NA          NA         NA        NA        NA
[2,] 0.1809942        NA          NA         NA        NA        NA
[3,] 0.4942798 0.3440664          NA         NA        NA        NA
[4,] 0.4062179 0.2430969 -0.11018558         NA        NA        NA
[5,] 0.4495496 0.2923421 -0.05750886 0.05301264        NA        NA
[6,] 0.8970190 0.8548073  0.72354215 0.77216750 0.7498497        NA
[7,] 0.9488000 0.9269964  0.85592633 0.88284975 0.8705822 0.3477377
[8,] 0.9509982 0.9300971  0.86182916 0.88771623 0.8759247 0.3673642
           [,7] [,8]
[1,]         NA   NA
[2,]         NA   NA
[3,]         NA   NA
[4,]         NA   NA
[5,]         NA   NA
[6,]         NA   NA
[7,]         NA   NA
[8,] 0.02250085   NA
      wavelength of length = 8
      fwhm for each wavelength
Call: cor.test(NA NA NA)

Models contain following parameters:
[[1]] p.value
[[2]] estimate
Dimension of each parameter: 8, 8, 1
```

```
History of usage
---------------------
(1)   Reflectance = mean applied to matrix spectra by attribute 'site'
(2)   Integrated spectra to WorldView2-8 channels
(3)   NRI values calculated
```

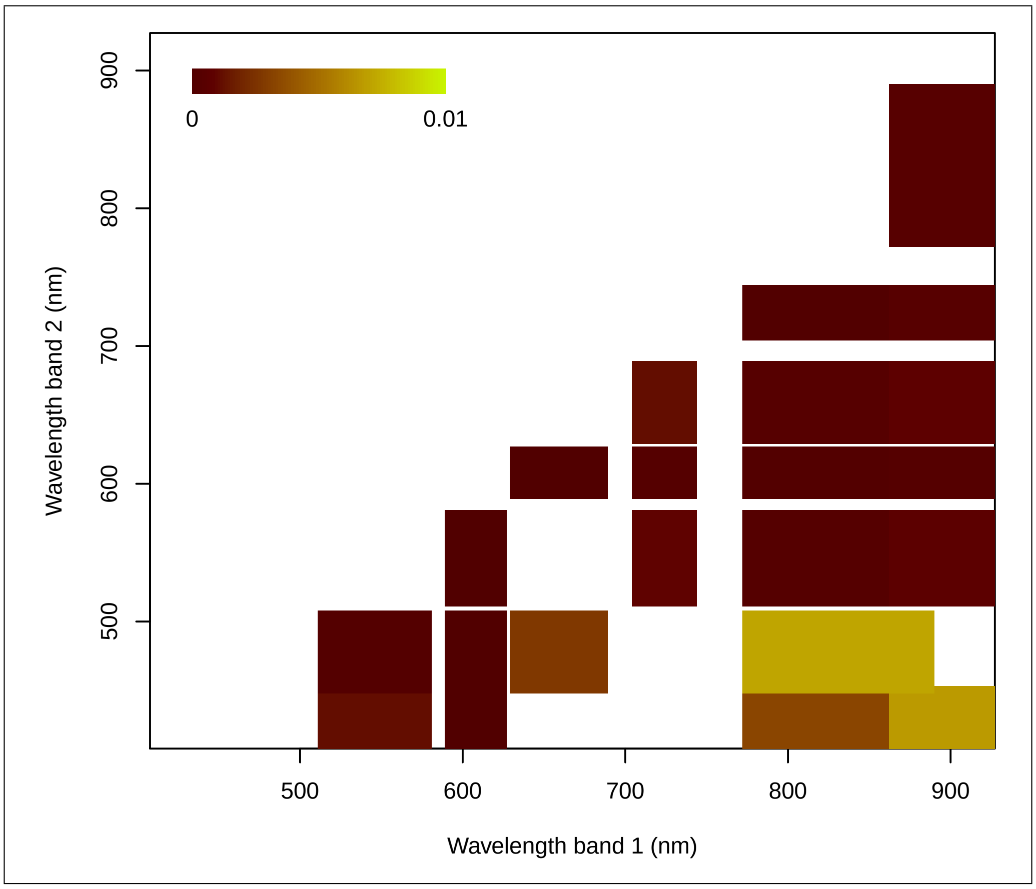
As you can see, there are p values and estimates of the correlation stored for each band combination. The coefficients of the correlation can be visualized by the function by plotting the object. The p-value is in most cases the interesting coefficient:

```
> plot(cortestnri, coefficient = "p.value")
```



Now it becomes obvious that the NRI from the band combination 4 and 3 is the definitely not correlated to vegetation cover. To see which NRI are significantly correlated let's only plot the p-values where NRI was correlated with a p value less than 0.01:

```
> plot(cortestnri, coefficient = "p.value", range = c(0,0.01))
```



Obviously all other NRI except the mentioned NRI from the bands 4 and 3 are significantly correlated to vegetation cover.

9.2 Linear models

Linear regressions between NRI and environmental variables can be performed using the ”lm.nri” function. See how this function works:

```
> str(lm.nri)

function (formula, preddata = NULL, ...)
```

The function requires the formula of the model as well as ”preddata” which is a Speclib or a data.frame containing the environmental variables. Use the Speclib and NRI data created in section 9 to perform a linear regression between NRI and fraction of vegetation which is stored as attribute in ”spec_WV”:

```
> lmnri <- lm.nri(nri_WV ~ chlorophyll, preddata = spec_WV)
```

See how the lmnri object looks like:

```
> lmnri

Data: nri, dimension: 8, 8, 45
          [,1]      [,2]        [,3]        [,4]      [,5]      [,6]
[1,]        NA        NA          NA          NA        NA        NA
[2,] 0.2002899        NA          NA          NA        NA        NA
[3,] 0.5084916 0.3431500          NA          NA        NA        NA
[4,] 0.4606805 0.2868589 -0.06243717          NA        NA        NA
[5,] 0.4664228 0.2935569 -0.05514847 0.007313881        NA        NA
[6,] 0.8562469 0.7917383  0.61592603 0.653241754 0.6490287        NA
[7,] 0.9133002 0.8726372  0.75581145 0.781375051 0.7785102 0.2617243
[8,] 0.9174453 0.8786030  0.76656735 0.791138887 0.7883868 0.2853859
           [,7] [,8]
[1,]         NA   NA
[2,]         NA   NA
[3,]         NA   NA
[4,]         NA   NA
[5,]         NA   NA
[6,]         NA   NA
[7,]         NA   NA
[8,] 0.02557163   NA

    ... (43 layers omitted)

          [,1]      [,2]        [,3]       [,4]      [,5]      [,6]
[1,]        NA        NA          NA         NA        NA        NA
[2,] 0.1809942        NA          NA         NA        NA        NA
[3,] 0.4942798 0.3440664          NA         NA        NA        NA
[4,] 0.4062179 0.2430969 -0.11018558         NA        NA        NA
[5,] 0.4495496 0.2923421 -0.05750886 0.05301264        NA        NA
[6,] 0.8970190 0.8548073  0.72354215 0.77216750 0.7498497        NA
[7,] 0.9488000 0.9269964  0.85592633 0.88284975 0.8705822 0.3477377
[8,] 0.9509982 0.9300971  0.86182916 0.88771623 0.8759247 0.3673642
           [,7] [,8]
[1,]         NA   NA
[2,]         NA   NA
[3,]         NA   NA
[4,]         NA   NA
[5,]         NA   NA
[6,]         NA   NA
[7,]         NA   NA
[8,] 0.02250085   NA
      wavelength of length = 8
      fwhm for each wavelength
Call: lm(nri_WV ~ chlorophyll)

Models contain following parameters:
[[1]] estimate
[[2]] std.error
[[3]] t.value
[[4]] p.value
[[5]] r.squared
```

```
Dimension of each parameter: 8, 8, 2


History of usage
---------------------
(1)   Reflectance = mean applied to matrix spectra by attribute 'site'
(2)   Integrated spectra to WorldView2-8 channels
(3)   NRI values calculated
```

Each model contains the parameters known from common linear regressions. However, the dimensions of the parameters make clear that there are 8*8 models stored in the object for which the parameters are available. Imagine you have even more bands than 8 you will most likely want to find out which is the best performing model. You can do this using the function ”nri_best_performance”:

```
> str(nri_best_performance)

function (nri, n = 1, coefficient = "p.value", predictor = 2,
    abs = FALSE, findMax = FALSE, ...)
```

The function takes the NRI data and the linear (or generalized linear) model as input. Further ”n” can be specified which is the number of best models which should be returned. The other parameters are not interesting at the moment.

In this example we want to get only the best model (n=1):

```
> nribest <- nri_best_performance(lmnri, n = 1)
> nribest

$Indices
  Band_1 Band_2
1    608    478

$Models

Call:
lm(formula = formula, data = glm_data)

Coefficients:
(Intercept)  chlorophyll
     0.3440      -0.0022
```

Maybe it it interesting to see the NRI values of the best performing NRI. Use ”getNRI” and the NRI data as well as the best performing NRI as input. For each sample of the Speclib, the NRI value of the best model is then shown:

```
> getNRI(nri_WV, nribest)

 [1] 0.2868589 0.3000261 0.2811214 0.3118467 0.2994424 0.2981061
 [7] 0.3026912 0.2501619 0.3055730 0.3027269 0.3077740 0.2571044
[13] 0.2899708 0.2831544 0.3031803 0.2373592 0.2618759 0.2484873
[19] 0.2557212 0.2597004 0.2599517 0.2698255 0.2547197 0.2666212
[25] 0.2652733 0.2653331 0.2612740 0.2595289 0.2557841 0.2412859
[31] 0.2521280 0.2256435 0.2620617 0.2560031 0.2326724 0.2524840
[37] 0.2566662 0.2510728 0.2627070 0.2551183 0.2452236 0.2529550
[43] 0.2293681 0.2546962 0.2430969
```

9.3 Generalized linear models

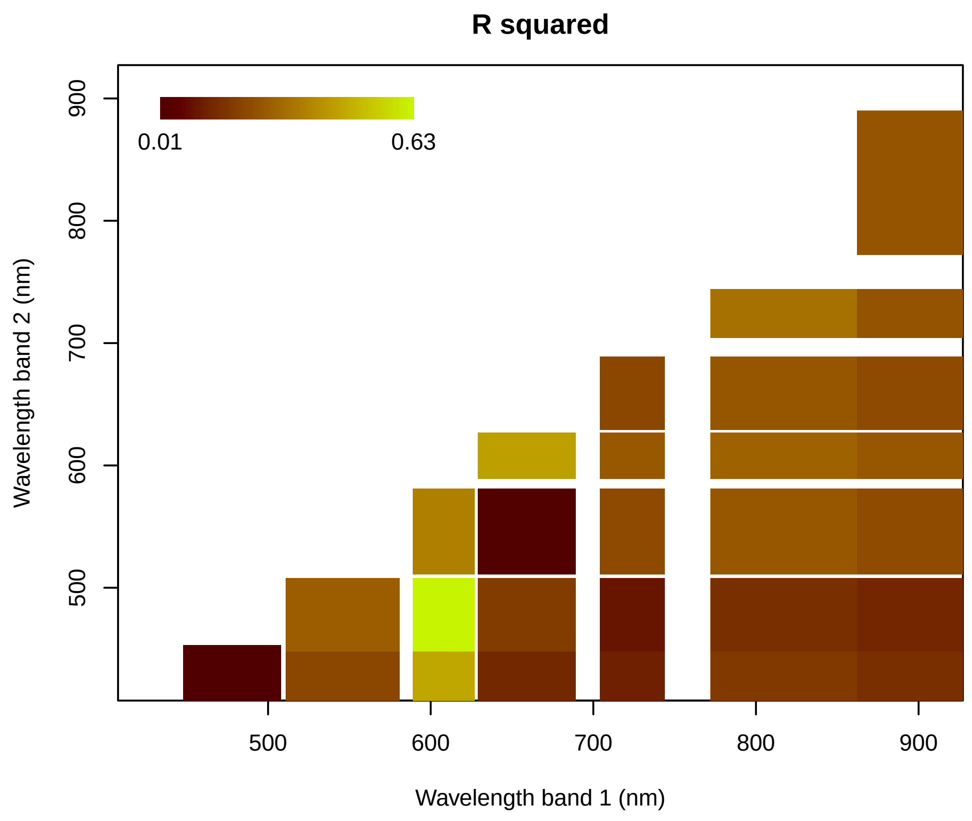
Calculation and plotting of generalized linear models work in the same way as the calculation of linear models in section 9.2. Note that the coefficients change because of the different models (e.g. r.squared is not available using glms).

9.4 Plot NRI models

Linear (or generalized linear) models of NRI and environmental variables can be plotted like shown in e.g. Meyer et al. (2013); Mutanga and Skidmore (2004b). Note: The plots in the cited studies based on NRI with narrow bands which were not resampled to e.g. WorldView channels like shown in this tutorial. You can easily create models and plots with narrow bands by omit the resampling in the beginning of this section 9 and using ”spectral_data” instead of the resampled data ”spec_WV”.

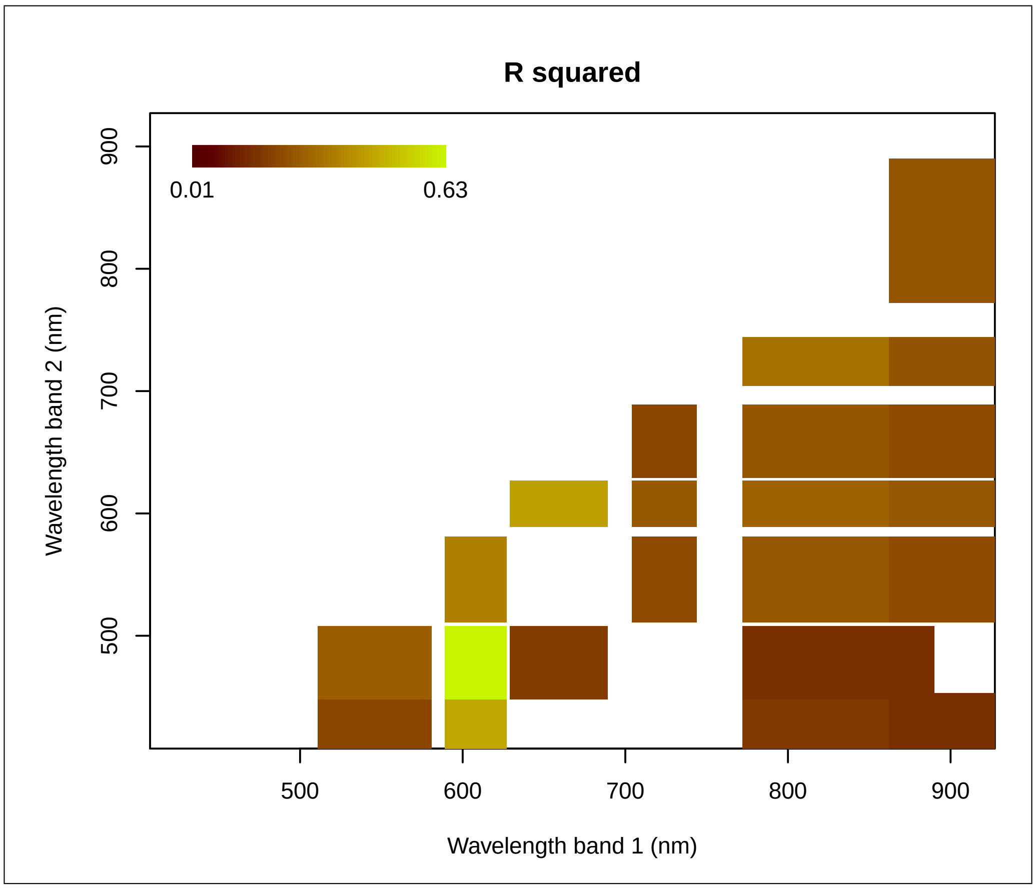
The plot.lmnri function takes a model from NRI and predictor variables (see section 9.2) and the coefficient to be plotted as input. Start with plotting the r.squared values of the linear model from section 9.2:

```
> plot(lmnri, coefficient = "r.squared", main = "R squared")
```



For each band combination, the r.squared value of the model of NRI and the environmental variable is represented by colour. Maybe you want to limit your plot to only these band combinations whose NRI were significantly related to the environmental variable. You can do this using the ”constraint” parameter. Assign your constraint as string. For example: constraint = ”p.value<0.01” means that only r.squared values of models with p value less than 0.01 will be drawn:

```
> plot(lmnri, coefficient = "r.squared", main = "R squared",
+      constraint = "p.value<0.01")
```



10 Linear spectral unmixing

Linear spectral unmixing is a method to derive the cover fractions of different materials within the footprint of multi- or hyperspectral pixels/measurements. For a detailed overview of linear spectral unmixing see e.g. Sohn and McCoy (1997). The algorithm in the hsdar package uses the code originally developed for Grass GIS by Markus Neteler. The basic concept behind linear spectral unmixing is that you provide a set of spectra (taken by a field spectrometer or multi/hyperspectral satellite sensor) and a set of spectra providing the information about the spectral properties of the pure materials which are mixed in the first set. The latter set is usually called "endmembers". Here, we will use two spectra from the USGS and define the endmember "vegetation" and "soil". The spectral to be unmixed are generated with PROSAIL:

```
> ## Use PROSAIL to generate some vegetation spectra with different LAI
> parameter <- data.frame(LAI = seq(0, 1, 0.01))
> spectral_data <- PROSAIL(parameterList = parameter)
> ## We resample the data to Quickbird channels to get the same
> ## spectral ranges
> spectral_data_qb <- spectralResampling(spectral_data, "Quickbird")
```

Now, we download the required endmember spectra from USGS's ftp-server.

```
> ## Get endmember spectra
> ## Retrieve all available spectra
> avl <- USGS_get_available_files()
> ## Download all spectra matching "grass-fescue"
> grass_spectra <- USGS_retrieve_files(avl = avl,
+                                      pattern = "grass-fescue")
> limestone <- USGS_retrieve_files(avl = avl, pattern = "limestone")
> ## Perform resampling for the endmember spectra. Note that we only
> ## use the first vegetation spectrum
> grass_spectra_qb <- spectralResampling(grass_spectra[1,],
+                                        "Quickbird")
> limestone_qb <- spectralResampling(limestone, "Quickbird")
```

Now, we merge the endmember spectra into one Speclib (and make sure that the range of the spectra is in [0,1]) and finally start the unmixing approach:

```
> em <- speclib(spectra = rbind(spectra(grass_spectra_qb),
+                               spectra(limestone_qb))/100,
+               wavelength = wavelength(limestone_qb))
> unmix_res <- unmix(spectral_data_qb, em)
> ## Let's have a look at the output:
> str(unmix_res)

List of 2
 $ fractions: num [1:2, 1:101] 0.895 0.104 0.897 0.103 0.898 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:2] "1" "2"
  .. ..$ : chr [1:101] "1" "2" "3" "4" ...
 $ error    : num [1:101] 0.0235 0.0234 0.0232 0.0231 0.023 ...
```

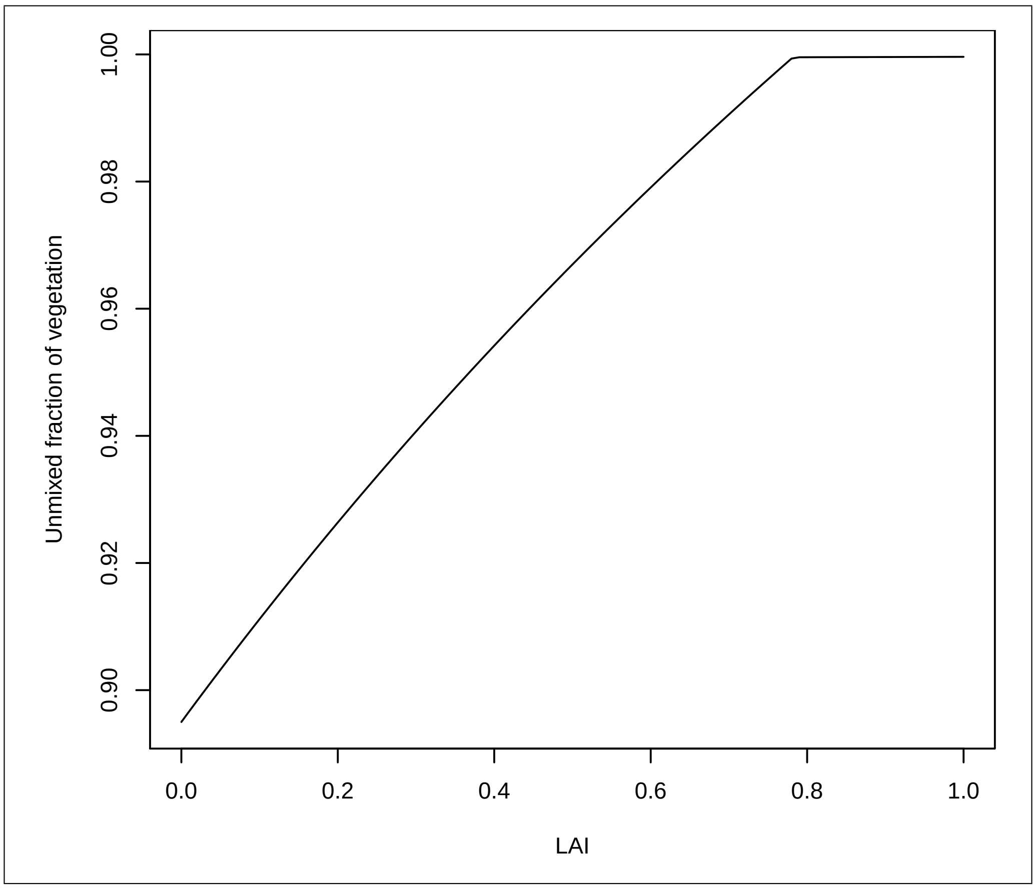
The return value of ”unmix” is a list with two elements:

1. ”fractions”: A matrix with the unmixed fractions of each endmember in each spectrum. The different spectra are the columns and the endmembers the rows of the matrix.

2. ”error”: Mathematically speaking, an over-determined linear equation system is solved during linear spectral unmixing. Thus, one is only able to minimize the error during solving the system. This amount of error is returned as the euclidean norm of the error vector after least square error minimisation. Large error values may indicate that endmember spectra do not fit well to the mixed material spectra.

Finally, we can generate a simple plot to visualize our results.

```
> plot(unmix_res$fractions[1,] ~ attribute(spectral_data_qb)$LAI,
+      type = "l", xlab = "LAI",
+      ylab = "Unmixed fraction of vegetation")
```


Unmixed fraction of vegetation
LAI
0.90
0.92
0.94
0.96
0.98
1.00
0.0
0.2
0.4
0.6
0.8
1.0

References

Bach, H., 1995. Die Bestimmung hydrologischer und landwirtschaftlicher Oberflächenparameter aus hyperspektralen Fernerkundungsdaten. Münchner Geographische Abhandlungen Reihe B, Band B21.

Clark, R. N., King, T. V. V., Gorelick, N. S., 1987. Automatic continuum analysis of reflectance spectra. In: Proceedings of the Third Airborne Imaging Spectrometer Data Analysis Workshop. pp. 138–142.

Jacquemoud, S., Baret, F., 1990. Prospect - a model of leaf optical-properties spectra. Remote Sensing of Environment 34 (2), 75–91.

Jacquemoud, S. A., Verhoef, W., Baret, F., Bacour, C., Zarco-Tejada, P. J., Asner, G. P., Francois, C., Ustin, S. L., 2009. PROSPECT + SAIL models: A review of use for vegetation characterization. Remote Sensing of Environment 113, Supplement 1 (0), 56 – 66.

Lehnert, L. W., Meyer, H., Meyer, N., Reudenbach, C., Bendix, J., 2013. Assessing pasture quality and degradation status using hyperspectral imaging: a case study from western Tibet. Vol. 8887. pp. 88870I–88870I–13.
URL http://dx.doi.org/10.1117/12.2028348

Lehnert, L. W., Meyer, H., Meyer, N., Reudenbach, C., Bendix, J., 2014. A hyperspectral indicator system for rangeland degradation on the Tibetan Plateau: A case study towards spaceborne monitoring. Ecological Indicators 39 (0), 54 – 64.

Lehnert, L. W., Meyer, H., Wang, Y., Miehe, G., Thies, B., Reudenbach, C., Bendix, J., Jul. 2015. Retrieval of grassland plant coverage on the Tibetan Plateau based on a multi-scale, multi-sensor and multi-method approach. Remote Sensing of Environment 164, 197–207.

Meyer, H., Lehnert, L. W., Wang, Y., Reudenbach, C., Bendix, J., 2013. Measuring pasture degradation on the Qinghai-Tibet Plateau using hyperspectral dissimilarities and indices. Vol. 8893. pp. 88931F–88931F–13.
URL http://dx.doi.org/10.1117/12.2028762

Mutanga, O., Skidmore, A., 2004a. Hyperspectral band depth analysis for a better estimation of grass biomass (*Cenchrus ciliaris*) measured under controlled laboratory conditions. International Journal of applied Earth Observation and Geoinformation 5 (2), 87–96.

Mutanga, O., Skidmore, A. K., 2004b. Narrow band vegetation indices overcome the saturation problem in biomass estimation. International Journal of Remote Sensing 25 (19), 3999–4014.

Sohn, Y. S., McCoy, R. M., 1997. Mapping desert shrub rangeland using spectral unmixing and modeling spectral mixtures with TM data. Photogrammetric Engineering and Remote Sensing 63 (6), 707–716.